Multi-domain, Higher Order Level Set Scheme for 3D Image Segmentation on the GPU

Ojaswa Sharma [1] os@imm.dtu.dk
Qin Zhang[2] zqyork@ices.utexas.edu
François Anton [1] fa@imm.dtu.dk
Chandrajit Bajaj [2] bajaj@cs.utexas.edu

[1] DTU Informatics
The Technical University of Denmark, Denmark

[2] Computational Visualization Center
The University of Texas at Austin, USA

Abstract

Level set method based segmentation provides an efficient tool for topological and geometrical shape handling. Conventional level set surfaces are only C^0 continuous since the level set evolution involves linear interpolation to compute derivatives. Bajaj et al. present a higher order method to evaluate level set surfaces that are C^2 continuous, but are slow due to high computational burden. In this paper, we provide a higher order GPU based solver for fast and efficient segmentation of large volumetric images. We also extend the higher order method to multi-domain segmentation. Our streaming solver is efficient in memory usage.

1. Introduction

The segmentation problem addressed here 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 $\cup\Omega_i = \Omega$, 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 (or level sets of functions on $\mathbb{R}^3$) naturally capture the arbitrary topology of the boundary of subdomains (Ω_i) in contrast to explicit or parameterized surfaces. Deformable level set surfaces under mean curvature flow (by solving a partial differential equation that minimizes some sort of an energy functional) provide a direct solution methodology for the segmentation problem. Pioneering work by Osher and Sethian [8] present an effective implicit representation for evolving curves. Later, the work was developed in context of the Mumford-Shah functional by Chan and Vese [5] for 2D images that do not contain prominent edges. In a subsequent paper, the 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 [10, 15].

Conventional level set methods solve the interface evolution equation with linear interpolation of the implicit function and its derivatives (at sampled grid points). The level set surface from this results in a C^0 continuous surface. Bajaj et al. [1, 2] present a tri-cubic B-spline based level set method that produces a C^2 continuous level set surface solution. 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 the most appropriate . In particular, in this paper we explore the multi-core parallelism of Graphics Processing Units (GPU's). 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 [9]. Of particular interest is a solution to the level set equations for segmenting large volumes. Work by [12, 7] show GPU based segmentation and visualization. Lefohn et al. [7] demonstrated an efficient sparse GPU segmentation using linear 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 [7] 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 sub-domains (or *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. Our solver operates in a streaming fashion and makes optimal use of the available GPU memory. Global algorithms cannot be directly applied

in a streaming compute environment. We employ streaming counterparts of various algorithms and show memory optimizations via a *host memory manager*. 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. Background

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 Ω 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 [8].

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, ϕ is generally chosen to be a signed distance function for certain properties that they offer [8, 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. In this paper, we show how the modified Mumford-Shah functional of Chan and Vese [5] can be minimized in a higher order fashion and also used to perform a multi-domain segmentation. 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 Ω, denoting $\Gamma^+ = \{(x,y,z) : \phi > 0\}$ as the interior of the volume bounded by Γ and $\Gamma^- = \{(x,y,z) : \phi < 0\}$ as the exterior of the volume bounded by Γ. A modified Mumford-Shah energy functional with regularization can be written as:

$$F(c_1,c_2,\Gamma) = \mu \cdot \text{Area}(\Gamma) + \nu \cdot \text{Vol}(\Gamma^+) + \lambda_1 \int_{\Gamma^+} |I-c_1|^2 dV + \lambda_2 \int_{\Gamma^-} |I-c_2|^2 dV, \quad (1)$$

where $\mu \geq 0, \nu \geq 0, \lambda_1 > 0$, and $\lambda_2 > 0$ are fixed scalar control parameters, and c_1 and c_2 are averages in Γ^+ and Γ^- respectively. A time varying variational form of equation (1) is:

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

where δ_ϵ is the smooth version of Dirac delta function. Bajaj et al. [1] propose to solve the higher order regularizing term in equation (2) by tri-cubic B-spline interpolation to compute accurate higher order derivatives of ϕ.

2.1. Multi-domain segmentation

Equation (2) defines two regions with respect to the zero level set surface of ϕ, 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 ϕ 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 Ω. Equation (2) is replaced by a system of m PDEs. We compactly write this system as:

$$\frac{\partial \phi_i}{\partial t} = \delta_\epsilon(\phi_i)\left\{\mu \nabla \cdot \left(\frac{\nabla \phi_i}{|\nabla \phi_i|}\right) - \nu - \sum_{k=0}^{2^{m-1}-1} \left[\left(\lambda_1 (I-c^1_{i,k})^2 - \lambda_2 (I-c^0_{i,k})^2\right) \prod_{p=0}^{m-1,\, p\neq i} \left(b_{k,p} + (-1)^{b_{k,p}} H_\epsilon(\phi_p)\right)\right]\right\}, \quad (3)$$

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

$$b_{k,p} = p^{th} \text{ bit in binary representation of } k, \in \{0,1\},$$

$$c^0_{i,k} = \text{mean}(I) \text{ in } (x,y,z) : \begin{cases} \phi_p > 0, \text{ if } b_{q,p} = 1, \\ \phi_p < 0, \text{ if } b_{q,p} = 0 \end{cases} \text{ with } q = 2k - k \bmod 2^i, \forall p \in [0, m-1], \quad (4)$$

$$c^1_{i,k} = \text{mean}(I) \text{ in } (x,y,z) : \begin{cases} \phi_p > 0, \text{ if } b_{q,p} = 1, \\ \phi_p < 0, \text{ if } b_{q,p} = 0 \end{cases} \text{ with } q = 2k + 2^i - k \bmod 2^i, \forall p \in [0, m-1]. \quad (5)$$

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

$$q = (k - (k \wedge (2^i - 1))) << 1 + k \wedge (2^i - 1), \text{and}$$
$$q = (k - (k \wedge (2^i - 1))) << 1 + 2^i + k \wedge (2^i - 1)$$

respectively. We find that this representation for keeping track of various regions of Φ is not only compact but also efficient for implementation on a constrained compute environment like the GPU.

3. Solver framework

We provide a framework for streaming computation of the solution to involved differential equations (2 or 3) based

on NVIDIA's parallel compute framework called CUDA [11]. The complexity of the solver is increased by the fact that m simultaneous PDEs need to be solved to arrive at a solution to equation (3). Our solver has a very small memory footprint on the *device (GPU)* compared to the actual volume that it can handle. On the *host* (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.

3.1. Host Memory Manager

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 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. We create a logical hierarchy by creating a 3D grid and a 3D block of voxels and assigning each voxel with a thread to process it. A linear grid of blocks is logically mapped to a 3D grid of blocks. For a 128^3 computational volume, a typical grid size could be 16^3 blocks with each block consisting 8^3 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 Figure 1.

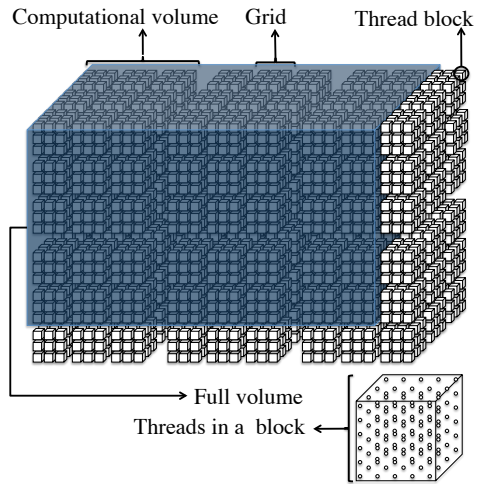


Figure 1. Volume hierarchy for spawning CUDA threads.

For operations that involve accessing voxel neighbors (e.g., finite differencing or convolution filtering), the host memory manager appropriately pads the computational volume to enable required number of shared voxels around the border of the computational volume. This effectively reduces the size of the volume to incorporate neighbors 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. Figure 2 shows a sketch of the setup.

Current sub-volumes from the image, the implicit functions and cubic coefficients are first transferred to the GPU memory as 3D textures. 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 volumes. Details of PDE evolution algorithms and respective kernels are presented in the next section.

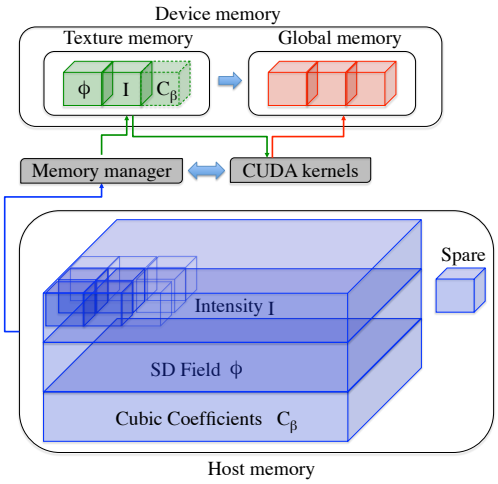


Figure 2. The memory manager.

4. Level set evolution

Following is our general approach in solving (3):

1. Interface initialization.
2. Signed distance field computation.
3. Average values computation.
4. Cubic coefficients computation.
5. PDE time stepping.
6. Reinitialization.
7. Repeat steps (3) - (6) until convergence.

8. Level set extraction and domain labeling.

Majority of these steps can be executed in a streaming fashion with an exception of average value computation.

4.1. Interface initialization

We choose to initialize the level set interface Γ to a bounding box or to a super-ellipsoid for a two domain segmentation. Multi-domain initialization should ensure that all the possible classes occupy non-zero region in space at the start so that all the domains have a scope of evolution. The domain Ω is partitioned into smaller sub-domains and each sub-domain is assigned small super-ellipsoids (with randomized centers) that form the interface Γ for each implicit function. Vese and Chan [14] observe a better and faster convergence in a 2D case for an initialization of the later kind. Our tests confirm this observation for the 3D case.

The kernel module for interface initialization computes the implicit function Φ 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.

4.2. Signed distance field

We create a narrow band signed distance field on GPU using the d-pass approach by Sharma and Anton [11] 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 [4] that uses optimal values of coefficients for distance multipliers to minimize accuracy error with an actual distance field. The algorithm has complexity $O(dN)$, where N is the total number of voxels in the volume.

The distance field is constructed layer by layer until the d layers are formed, making a narrow band of width $2d$. Every voxel is updated based on the values of the neighbors. 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, thus updating 126^3 voxels in a computational volume.

4.3. Average values

In a two-domain segmentation, the zero level set of ϕ divides Ω 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 subsection 2.1) 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 [3]. A CUDA implementation of the same exists as the CUDPP library by Harris and Sengupta [6]. 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 fast CPU computation of the averages.

4.4. Cubic coefficients

Let

$$\beta^3(x) = \begin{cases} \frac{2}{3} - x^2 + \frac{1}{2}|x|^3, & 0 \le |x| < 1, \\ \frac{1}{6}(2-|x|)^3, & 1 \le |x| < 2, \\ 0, & 2 \le |x|, \end{cases}$$

be the univariate cubic B-spline basis function over the knots $\{-2,-1,0,1,2\}$, and let $s(x) = \sum_{i=1}^{n-1} c_i \beta^3(x-i)$ be a cubic spline function. Then the spline interpolation problem is to determine the coefficients $\mathbf{c} = \{c_i\}_{i=1}^{n-1}$ such that the following interpolation conditions

$$s(k) = \sum_{i=1}^{n-1} c_i \beta^3(k-i), \;\; k = 1, \cdots, n-1 \qquad (6)$$

are satisfied for any given function values $\{s(k)\}_{k=1}^{n-1}$. To avoid inversion of a large matrix to solve the linear system (6), a better approach is to compute these coefficients in a recursive fashion (see [1]).

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 computing cubic coefficients along each direction sequentially. These correspond to tensor product splines in $\mathbb{R}^3$. Rather than operating in the computational volume, we derive coefficients along an axis in planar sections orthogonal to one of the coordinate axes. The computation takes place in three steps, sequentially along each coordinate direction.

The host memory manager determines the largest possible volume slice (*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.

4.5. PDE time stepping

The PDEs are solved by discretizing equation (3) to compute 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. We again use binary indexing 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), we need cubic coefficients for all the implicit functions in the device memory 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 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, we try to minimize the unusable device memory locked in the texture 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, we try to make m_x as large as possible, followed by a similar heuristic for m_y.

With sub-volumes of Φ, I and the coefficients cached on the GPU, PDE update is computed for every voxel and for all the implicit functions.

4.6. Results

We present the results of our multi-domain segmentation on a CT (Computed Tomography) volume of a human thoracic region, 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. The program has been implemented in the freely available software UT-CVC image processing and visualization software called VolRover [13].

Figure 3 shows a volume visualization of our human thoracic region dataset. The CT 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 4. The segmentation yields four prominent classes. These classes separate regions of ribs, spinal chord, lungs and bronchioles as shown in Figure 4(a), (b), (c), and (d) respectively.

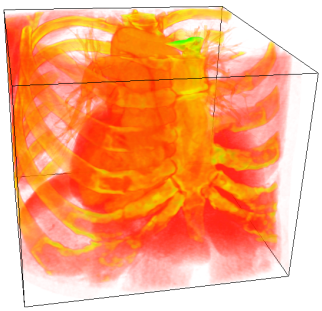
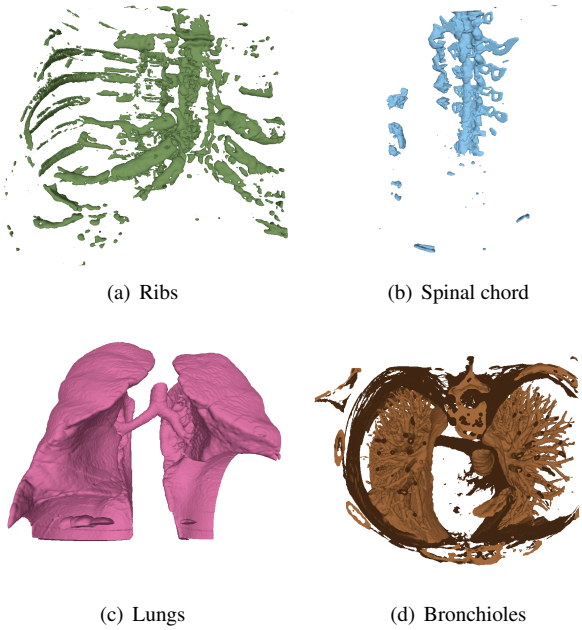


Figure 3. Volume visualization of computed Tomography (CT) volume of a human thoracic region dataset.



(a) Ribs (b) Spinal chord

(c) Lungs (d) Bronchioles

Figure 4. Multi-domain segmentation of the CT volume into 4 classes.

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 tri-linear uses finite

differencing to compute double derivatives, while the tri-cubic uses texture lookups and fewer computations. Tri-cubic B-spline coefficients are expensive to compute and we obtain a speedup of about 3x here.

Performance speedups of GPU computations compared to CPU ones are shown in Figures 5 and 6. 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. $1602 \times 1125 \times 195$ 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.

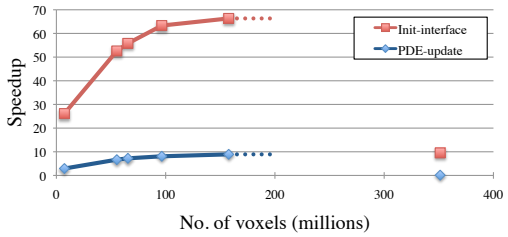


Figure 5. GPU speedup for tri-linear level set segmentation

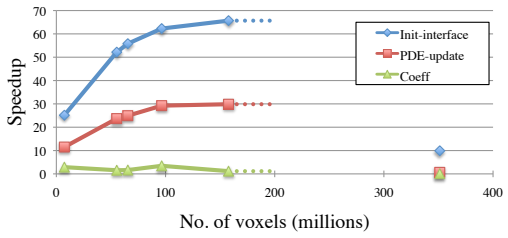


Figure 6. GPU speedup for tri-cubic level set segmentation

5. Conclusions

In this work we present an efficient parallel (multi-core) CUDA computation of a multi-domain, higher order level set method applied to a Mumford-Shah like energy functional. The higher order framework can be easily used with other energy functional as well. We show results of the segmentation on CT volumes along with performance speedups obtained with our GPU based implementation. The overall performance gains we obtain is 20x for the two-domain segmentation and 10x for the multi-domain segmentation.

Acknowledgement

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. This work was done while Ojaswa Sharma was visiting Chandrajit Bajaj at UT-CVC. His visit was supported by the Technical University of Denmark.

References

[1] C. Bajaj, G. Xu, and Q. Zhang. A Higher Order Level Set Method with Applications to Smooth Surface Constructions. ICES Report 06-18. *Institute for Computational Engineering and Sciences, The University of Texas at Austin*, 2006.

[2] C. Bajaj, G. Xu, and Q. Zhang. A fast variational method for the construction of resolution adaptive C2-smooth molecular surfaces. *Computer Methods in Applied Mechanics and Engineering*, 198(21-26):1684–1690, 2009.

[3] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[4] G. Borgefors. Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing*, 34(3):344–371, 1986.

[5] T. F. Chan and L. A. Vese. A level set algorithm for minimizing the Mumford-Shah functional in image processing. In *IEEE Workshop on Variational and Level Set Methods*, pages 161–168, 2001.

[6] M. Harris, S. Sengupta, and J. Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, 3, 2007.

[7] A. Lefohn, J. Kniss, C. Hansen, and R. Whitaker. A streaming narrow-band algorithm: interactive computation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):422–433, 2004.

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

[9] G. Papandreou and P. Maragos. Multigrid geometric active contour models. *IEEE Transactions on Image Processing*, 16(1):229, 2007.

[10] L. Rudin, S. Osher, and E. Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D*, 60:259–268, 1992.

[11] 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*, 2009.

[12] R. Strzodka and M. Rumpf. Level set segmentation in graphics hardware. In *Proc. IEEE International Conference on Image Processing (ICIP-2001)*, pages 1103–1106.

[13] UT-CVC. Volume Rover, 2006. `http://cvcweb.ices.utexas.edu/cvc/projects/project.php?proID=9`.

[14] L. A. Vese and T. F. Chan. A multiphase level set framework for image segmentation using the Mumford and Shah model. *International Journal of Computer Vision*, 50(3):271–293, 2002.

[15] J. Weickert, B. Romeny, M. Viergever, et al. Efficient and reliable schemes for nonlinear diffusion filtering. *IEEE Transactions on Image Processing*, 7(3):398–410, 1998.