Parallel fuzzy connected image segmentation on GPU

Ying Zhuge[a)]
Radiation Oncology Branch, National Cancer Institute, National Institutes of Health, Bethesda, Maryland 20892

Yong Cao
Computer Science Department, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24060

Jayaram K. Udupa
Medical Image Processing Group, Department of Radiology, University of Pennsylvania, Philadelphia, Pennsylvania 19104

Robert W. Miller
Radiation Oncology Branch, National Cancer Institute, National Institutes of Health, Bethesda, Maryland 20892

(Received 30 March 2011; revised 13 May 2011; accepted for publication 20 May 2011; published 30 June 2011)

Purpose: Image segmentation techniques using fuzzy connectedness (FC) principles have shown their effectiveness in segmenting a variety of objects in several large applications. However, one challenge in these algorithms has been their excessive computational requirements when processing large image datasets. Nowadays, commodity graphics hardware provides a highly parallel computing environment. In this paper, the authors present a parallel fuzzy connected image segmentation algorithm implementation on NVIDIA's compute unified device Architecture (CUDA) platform for segmenting medical image data sets.

Methods: In the FC algorithm, there are two major computational tasks: (i) computing the fuzzy affinity relations and (ii) computing the fuzzy connectedness relations. These two tasks are implemented as CUDA kernels and executed on GPU. A dramatic improvement in speed for both tasks is achieved as a result.

Results: Our experiments based on three data sets of small, medium, and large data size demonstrate the efficiency of the parallel algorithm, which achieves a speed-up factor of 24.4x, 18.1x, and 10.3x, correspondingly, for the three data sets on the NVIDIA Tesla C1060 over the implementation of the algorithm on CPU, and takes 0.25, 0.72, and 15.04 s, correspondingly, for the three data sets.

Conclusions: The authors developed a parallel algorithm of the widely used fuzzy connected image segmentation method on the NVIDIA GPUs, which are far more cost- and speed-effective than both cluster of workstations and multiprocessing systems. A near-interactive speed of segmentation has been achieved, even for the large data set. [DOI: 10.1118/1.3599725]

Key words: image segmentation, fuzzy connectedness, fuzzy affinity, GPU

I. INTRODUCTION

Image segmentation is the process of identifying and delineating objects in images. It is the most crucial of all computerized operations done on acquired images. Even seemingly unrelated operations, like image (gray-scale/color) display, 3D visualization, interpolation, filtering, and registration depend, to some extent, on image segmentation; since they all would need some object information for their optimum performance. In spite of several decades of research,[1,2] segmentation remains a challenging problem in both image processing and computer vision.

The fuzzy connectedness (FC) framework and its extensions[3–8] have been extensively utilized in many medical applications, including multiple sclerosis (MS) lesion detection and quantification via MR imaging,[9] upper airway segmentation in children via MRI for studying obstructive sleep apnea,[10] electron tomography segmentation,[11] abdominal segmentation,[12] automatic brain tissue segmentation[8] in MRI images, clutter-free volume rendering and artery–vein separation in MR angiography,[13] in brain tumor delineation via MR imaging,[14] and automatic breast density estimation via digitized mammograms for breast cancer risk assessment.[15] High segmentation accuracies of the FC algorithms have been achieved in these clinical applications, such as true positive volume fraction of more than 96% in brain tissue segmentation,[8] false negative volume fraction of 1.3% in MS lesion detection and quantification,[9] an accuracy of 97% in upper airway segmentation.[10] New imaging technologies are capable of producing increasingly larger image data sets, and newer and advanced applications in quantitative

radiology call for real-time or interactive segmentation methodologies. One problem with the FC image segmentation algorithms has been their high computational requirements for processing large image data sets.[16]

Several parallel implementations have been developed to improve the efficiency of the fuzzy connectedness algorithms. A parallel implementation of the scale-based FC algorithm has been developed for implementation on a cluster of workstations (COWs) by using the message passing interface (MPI) parallel-processing standard.[16] A manager-worker scheme has been used in this implementation. The manager processor keeps a watch on the worker processors as to whose queues become empty and who, therefore, may become idle. A worker processor may be activated because there are chunks (subsets) of image data, whose queues are still not empty. The entire segmentation process stops at the point when all queues maintained by all processors (the workers) become empty. A speed-up factor of approximately three has been achieved on a COW with six workstations. An OpenMP-based parallel implementation of the fuzzy connectedness algorithm has been reported recently.[17] This algorithm has been adapted to a distributed-processing scheme.[7] This scheme also works using manager-worker paradigm in which there are several processors (the workers) handling subsets of the data set and a special processor (the manager) that controls how the other processors carry out the segmentation of their corresponding subset. A speed increase of approximately five has been achieved related to the sequential implementation. OpenMP requires special compilers that recognize directives embedded in the source code which control parallelism. Typically, OpenMP systems are expensive, tightly coupled, shared memory multiprocessor systems (MPS), such as the SGI Altix 4700, which is the hardware being used in Ref. 17. The lower boost factor achieved in Ref. 16 compared to Ref. 17 might be due to the very fast FC algorithm employed[18] to begin with and due to the differences in image data, objects segmented, and the computing platforms used.

The aim of this study is to describe a parallel implementation of the FC algorithm using less expensive hardware, which can achieve vastly higher speed-up factors than both the COW and MPS systems. Toward this goal, we chose massively parallel graphics processing units (GPU) to accelerate the FC algorithms, mainly because the GPU has substantial arithmetic and memory bandwidth capabilities. Coupled with its recent addition of user programmability, it has permitted general-purpose computation on graphics hardware (GPGPU).[19] Moreover, the GPU has very low cost, is wide available, and can be used on a normal desktop PC.

The use of GPU as a hardware accelerator has attracted much recent attention and has proved to be an effective approach in the domain of high performance scientific computation.[20] Since medical imaging applications intrinsically have data-level parallelism with high compute requirements, they are highly suitable for implementation on the GPU. Several studies of medical imaging applications on GPU have been reported recently, such as in deformable image registration,[21] Monte Carlo-based dose calculation in radiotherapy,[22] and image segmentation.[23–26] An interactive, GPU-based level set segmentation tool called GIST has been developed for 3D medical images in Ref. 23. A sparse level set solver has been implemented on the GPU. An improvement of the GPUs narrow band algorithm has been proposed.[24] The communication latency between the GPU and CPU that exists[23] has been avoided by traversing the domain of active tiles in parallel on the GPU. Thus, the computational efficiency has been dramatically improved. A GPU-based random walker method has been presented for interactive organ segmentation in 2D and 3D medical images.[25] It is implemented on an ATI Radeon X800 XT graphics card by using a graph cuts interface. The computation time has been reduced in excess of an order of magnitude, compared with the CPU version. A fast graph cut method on the NVIDIA's GPU using the compute unified device architecture (CUDA) framework has also been proposed.[26] The performance of over 60 graph cuts per second on 1024×1024 2D images and over 150 graph cuts per second on 640×480 2D images on an Nvidia 8800 GTX have been reported.

Theoretical and algorithmic studies of the similarities among FC, level set, graph cut, and watershed families of methods are recently emerging,[27–29] which also demonstrate their differences in computational efficiencies. While comparative analysis of the GPU implementations of these frameworks will be worthwhile, their individual GPU implementations have to be developed first. Since such implementations do not exist for FC, and with this goal in mind, in this paper, we present a parallel FC image segmentation algorithm implemented on GPU by using CUDA to achieve nearly interactive speed when segmenting large medical image data sets. We emphasize that our focus in this paper is algorithmic speed. The accuracy of the parallel FC algorithm implemented on GPU is the same as that of the sequential algorithm on CPU. We do not undertake any application-specific evaluation of the parallel FC algorithm in this paper. In Sec. II, we first briefly present the basic FC principles and its sequential algorithm that is chosen for parallel GPU implementation; we then describe the NVIDIA GPU architecture and the CUDA programming model, and explain the parallelized version of this algorithm and its implementation by using CUDA. The experimental results are presented in Sec. III. Finally, a discussion and our concluding remarks are presented in Sec. IV. An early version of this paper was presented at the 31st Annual International Conference of the IEEE Engineering in Medicine and Biology Society, whose proceedings contained an abbreviated version of this paper.[30]

II. MATERIALS AND METHODS

II.A. Fuzzy connectedness principles and sequential algorithm

We chose to parallelize and implement on GPU the first FC segmentation algorithm reported in Ref. 3. Although several advanced versions of FC algorithms have been reported (see Ref. 31 for a recent review), such as scale-based FC, relative FC, and iterative relative FC, we chose the absolute FC of Ref. 3 since this algorithm forms the basic building block

of all other algorithms. The lessons learned from its implementation are more fundamental and can directly lead us to the realization of other algorithms on GPU.

The characteristics of both the algorithm and the GPU platform play a crucial role in determining the final speed-up factor. We briefly describe the concepts related to FC in this section to make this paper self-contained. Please see the original papers for more details.[3,4]

We refer to a volume image as a scene and represent it by a pair $\mathcal{C}=(C, f)$, where C is a rectangular array of cuboidal volume elements, usually referred to as voxels, and f is the scene intensity function which assigns to every voxel $c \in C$ an integer called the intensity of c in $\mathcal{C}$ in a range [L, H].

II.A.1. Fuzzy adjacency and affinity

Independent of any image data, we think of the digital grid system defined by the voxels as having a fuzzy adjacency relation. This relation assigns to every pair (c,d) of voxels a value between zero and one. The closer c and d are spatially to each other, the greater is this number. This is intended to be a "local" phenomenon. How local it ought to be should depend on the blurring property of the imaging device. We denote the fuzzy adjacency relation by α and the degree of adjacency assigned to any voxels (c,d) by $\mu_\alpha(c,d)$.

Now, consider the voxels as having scene intensities assigned to them. We define another local fuzzy relation called affinity on voxels denoted by κ. The strength of this relation between any voxels c and d, denoted $\mu_\kappa(c,d)$, lies between zero and one, and indicates how the voxels "hang together" locally in the scene. $\mu_\kappa(c,d)$ depends on $\mu_\alpha(c,d)$, as well as on how similar are the intensities or intensity-based properties at c and d. The properties of fuzzy affinity relations are studied extensively and a guidance as to how to set up fuzzy affinities in practical applications is given in Refs. 4 and 32. In this paper, the following functional form for μ_κ:

$$\mu_\kappa(c,d) = \mu_\alpha(c,d)\sqrt{g_1(f(c),f(d))g_2(f(c),f(d))}, \quad (1)$$

is used, where g_1 and g_2 are Gaussian functions of $[f(c)+f(d)]/2$ and $[f(c) - f(d)]/2$, respectively. The mean and variance of g_1 are related to the mean and variance of the intensity of the object that we wish to define in the scene. That is, this component of affinity takes on a high value when c and d are both close to an expected intensity value for the object. g_2 is a zero-mean Gaussian, the underlying idea being to capture the degree of local hanging togetherness of c and d based on intensity homogeneity.

II.A.2. Fuzzy connectedness and fuzzy objects

The aim in FC is to capture the global phenomenon of hanging togetherness in a global fuzzy relation on voxels called fuzzy connectedness, denoted κ. The strength of this relation $\mu_K(c,d)$ between any voxels c and d, indicating the strength of their connectedness, lies between zero and one, and is determined as follows: There are numerous possible "paths" within the scene domain C between c and d. Each path for our purposes is a sequence of voxels, starting from c and ending in d, with the successive voxels being nearby. We think of each pair of successive voxels as constituting a link and the whole path to be a chain of links. We assign a strength (between zero and one) to every path, which is simply the smallest pairwise voxel affinity along the path. Finally, the strength of connectedness between c and d is the strength associated with the strongest of all paths between c and d.

Let θ be any number in [0,1], a fuzzy connected object $\mathcal{O}$ in $\mathcal{C}$ of strength θ, and containing a voxel o, consists of a pool $O \subset C$ of voxels together with a value indicating "objectness" assigned to every voxel. O is such that $o \in O$, and for any voxels $c \in O$ and $d \in O$, the strength of connectedness between them $\mu_K(c,d) \geq \theta$, and for any voxel $c \in O$ and $e \notin O$, the strength $\mu_K(c,e) < \theta$.

The absolute FC algorithm is presented below for segmenting an object $\mathcal{O}$ in $\mathcal{C}$. For any voxel affinity κ in $\mathcal{C}=(C,f)$, we define the *κ-connectivity scene* of $\mathcal{C}$ with respect to a voxel $o \in C$ by $\mathcal{C}_{Ko}=(C,f_{Ko})$, where, for any $c \in C$, $f_{Ko}=\mu_K(o,c)$. The algorithm uses Dijkstra's implementation of dynamic programming to find the best path from o to each voxel in $\mathcal{C}$.

Algorithm κ FOE (Ref. 3)

Input: $\mathcal{C}=(C,f)$, any $o \in C$, fuzzy affinity κ.

Output: A κ-connectivity scene $\mathcal{C}_{Ko}=(C, f_{Ko})$ of $\mathcal{C}$ with respect to o.

Auxiliary Data Structures: 3D array representing the connectivity scene $\mathcal{C}_{Ko}=(C, f_{Ko})$ and a queue Q containing voxels to be processed. We refer to the array itself by $\mathcal{C}_{Ko}$ for the purpose of the algorithm.

Begin

1. set all voxels of $\mathcal{C}_{Ko}$ to 0 except o which is set to 1;
2. push o to Q;
3. **while** Q is not empty **do**
4. remove a voxel c from Q for which $f_{Ko}(c)$ is maximal;
5. **for** each voxel e such that $\mu_\kappa(c,e)>0$ **do**
6. set $f_{\min}=\min\{f_{Ko}(c), \mu_\kappa(c,e)\}$;
7. **if** $f_{\min}>f_{Ko}(e)$ **then**
8. set $f_{Ko}(e)=f_{\min}$;
9. **if** e is already in Q **then**
10. update the location of e in Q;
11. **else**
12. push (e) in Q;

End

II.B. Parallel implementation

In this section, we first briefly describe the NVIDIA GPU hardware architecture and the CUDA programming model. For a full description on NVIDIA GPU and CUDA, readers are referred to the CUDA programming guide.[33] We then describe how we implement the FC algorithm using CUDA.

II.B.1. NVIDIA GPU architecture and programming model

The underlying hardware architecture of a NVIDIA GPU is illustrated in Fig. 1. The NVIDIA Tesla C1060 GPU is

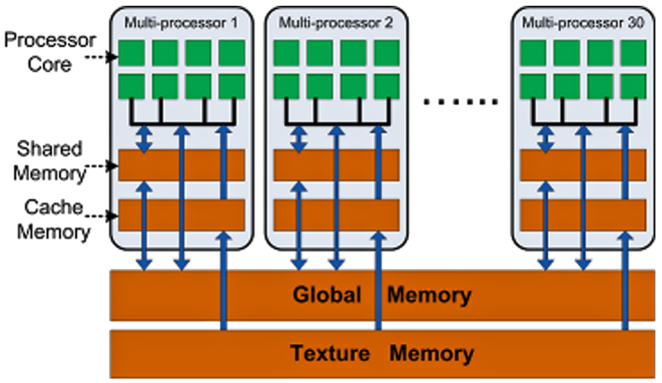

FIG. 1. NVIDIA GPU hardware architecture.

used as an example to provide a brief overview of the architecture. The Tesla C1060 GPU has 240 processing cores with a clock rate of 1.3 GHz for each core, delivering nearly one Tera FLOPS of computational power. To support an intuitive and flexible programming environment to access such computing power, NIVIDA provides CUDA framework,[33] which is based on a C-language model. CUDA enables generation and management of a massive number of processing threads, which can be executed in parallel on GPU cores with efficient hardware scheduling.

The 240 cores of Tesla C1060 GPU are grouped into 30 multiprocessors. Each multiprocessor has eight processing cores, organized in a SIMD (single instruction multiple data) fashion. Each core has its own register file and arithmetic logic unit, which allows it to accomplish a specific computational task. The Tesla C1060 has 4 GB of on-board device memory, which can be used as read-only texture memory or read-write global memory. The GPU device memory features have very high bandwidth, recorded at 102 GB s^{-1}, but it suffers from high access latency. In each multiprocessor unit, there is 16 KB of user-controlled L1 cache, called shared memory. If it is used efficiently, it can be employed to hide the latency in global memory access.

The CUDA programming model is based on concurrently executed threads. CUDA manages threads in a hierarchical structure. Threads are grouped into a thread block, and thread blocks are grouped into a grid. All threads in one grid share the same functionality, as they are executing the same kernel code. Each thread block is mapped on to one multiprocessor unit, and threads in each block are scheduled to run on eight processing cores of the multiprocessor unit, using a scheduling unit of 32-thread warp. Since the threads in a block are executed on the same multiprocessor, they can use the same shared memory space for data communication. On the other hand, the threads between different blocks can communicate only through the low-speed global memory.

II.B.2. CUDA implementation

In CUDA, programs are expressed as kernels. In order to map a sequential algorithm to the CUDA programming environment, developers should identify data-parallel portions of the application and isolate them as CUDA kernels. In the FC algorithm, there are two major computational tasks: (i) computing the fuzzy affinity relations and (ii) computing the fuzzy connectedness relations. We shall refer to (i) as "affinity computation" and (ii) as "tracking" a fuzzy object. These two tasks are implemented as CUDA kernels, and a dramatic improvement in speed for both tasks is achieved as a result.

(1) *Affinity computation kernel*: The CUDA implementation of fuzzy affinity computation is straightforward. The fuzzy affinity computation of every pair (c,d) of voxels where $\mu_\alpha(c,d)$ is greater than zero is totally independent of other pair of voxels. Thus for the pair (c,d), one thread is assigned to compute corresponding $g_1(c,d)$ and $g_2(c,d)$ in Eq. (1), and the fuzzy affinity $\mu_\kappa(c,d)$ result is written to the specific allocated GPU device memory. For computational simplicity, we use the six-adjacency relation for α.

(2) *Tracking kernel*: Computing the fuzzy connectedness values for a fuzzy object is a variation of the single-source-shortest-path (SSSP) problem. Dijkstra's algorithm is an optimal sequential solution to the SSSP problem. Parallel implementation of the Dijkstra's SSSP algorithm is quite challenging.[34] As far as is known, there is no efficient parallel algorithm for the SSSP problem in a SIMD model. Harish and Narayanan[35] proposed the use of CUDA to accelerate large graph algorithms (including those for the SSSP problem) on the GPU; however, they implemented only a very basic version and did not gain much performance improvement. We use a similar scheme, but take advantage of a newer version of CUDA hardware, which supports atomic read/write operations in the device global memory. The flow chart of our CUDA implementation is illustrated in Fig. 2, and the algorithm is presented below.

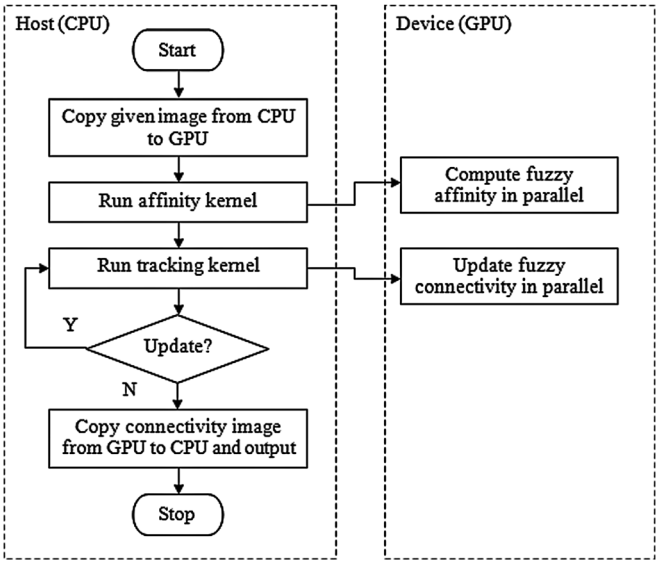

FIG. 2. The flow chart of the proposed CUDA implementation.

Algorithm CUDA-FOE

Input: $\mathcal{C}=(C,f)$, any $o\in C$, the functional form of κ.

Output: A κ-connectivity scene $\mathcal{C}_{Ko}=(C,f_{Ko})$ of $\mathcal{C}$ with respect to o.

Auxiliary Data Structures: Two 3D arrays representing binary scenes $\mathcal{C}_{\mathrm{m1}}=(C,f_{\mathrm{m1}})$ and $\mathcal{C}_{\mathrm{m2}}=(C,f_{\mathrm{m2}})$. We refer to the arrays themselves by $\mathcal{C}_{\mathrm{m1}}$ and $\mathcal{C}_{\mathrm{m2}}$ for the purpose of the algorithm.

Begin

1. set all voxels of $\mathcal{C}_{Ko}$ and $\mathcal{C}_{\mathrm{m1}}$ to 0 except o which is set to 1;
2. Invoke AFFINITY-KERNEL on grid to compute fuzzy affinity μ_κ.
3. **while** $\mathcal{C}_{\mathrm{m1}}$ is not all zero **do**
4. set all voxels of $\mathcal{C}_{\mathrm{m2}}$ to 0;
5. Invoke TRACKING-KERNEL($\mathcal{C}_{\mathrm{Ko}},\mathcal{C}_{\mathrm{m1}},\mathcal{C}_{\mathrm{m2}},\mu_\kappa$) on grid;
6. Copy $\mathcal{C}_{\mathrm{m2}}$ to $\mathcal{C}_{\mathrm{m1}}$;

End

The above algorithm is executed on the host (CPU) side while the kernels are executed on the device (GPU) side, as shown in Fig. 2. The different threads operate on voxels for updating connectivity information simultaneously. In one invocation, they all update connectivity information as much as they can on the voxels in their purview. The CPU determines if any updating has been done in the last invocation of the tracking kernel. If so, the kernel is invoked again with updated information on the voxels (in $\mathcal{C}_{\mathrm{m2}}$) where connectivity needs to be further updated. The CPU terminates the run of the algorithm where no more updates are presented by the tracking kernel. The affinity kernel and the tracking kernel algorithms are presented below.

AFFINITY-KERNEL

Begin

1. compute thread index t(id);
2. **for** each voxel c processed by t(id) **do**
3. **for** each voxel d adjacent to c **do**
4. compute affinity $\mu_\kappa(c,d)$ using Equation 1;
5. write $\mu_\kappa(c,d)$ to corresponding GPU memory;

End

TRACKING-KERNEL ($\mathcal{C}_{\mathrm{Ko}},\mathcal{C}_{\mathrm{m1}},\mathcal{C}_{\mathrm{m2}},\mu_\kappa$)

Begin

1. compute thread index t(id);
2. **for** each voxel c processed by t(id) **do**
3. **if** $\mathrm{f}_{\mathrm{m1}}(c)=1$ **then**
4. **for** each voxel e such that $\mu_\kappa(c,e)>0$ **do**
5. set $f_{\min}=\min\{f_{Ko}(c),\mu_\kappa(c,e)\}$;
6. **if** $f_{\min}>f_{Ko}(e)$ **then**
7. set $f_{Ko}(e)=\mathrm{f}_{\min}$;
8. set $f_{\mathrm{m2}}(e)=1$;

End

The algorithm CUDA-FOE is an iterative procedure. At the first iteration, only one thread which processes the voxel o is active. TRACKING-KERNEL is called to update the connectivity scene $\mathcal{C}_{Ko}$ and the binary scene $\mathcal{C}_{\mathrm{m2}}$. More threads will be involved and become active for the connectivity update at the successive iteration. Because of the limited communication capability among threads from different blocks, the CPU side needs to collect connectivity update information from each thread on GPU, and decides when to terminate calling the TRACKING-KERNEL. Each thread checks the flag of each voxel under its control to see if it is 1 in the binary array $\mathcal{C}_{\mathrm{m1}}$. If **yes** (which means the connectivity of that voxel has been updated during the last iteration, thus the connectivity values of its neighboring voxels might need to be updated in current iteration), it fetches its connectivity value $f_{Ko}(c)$ from the connectivity array $\mathcal{C}_{Ko}$ and the affinities $\mu_\kappa(c,e)$ between voxel c and its adjacent voxel e. Then the connectivity value of voxel e is updated if the minimum of $\mu_\kappa(c,e)$ and $f_{Ko}(c)$ is greater than its original connectivity value $f_{Ko}(e)$. Note in line 7 of the Algorithm TRACKING-KERNEL, atomic operation was used for consistency, because update operations for one voxel by multiple threads might happen simultaneously. The two binary arrays $\mathcal{C}_{\mathrm{m1}}$ and $\mathcal{C}_{\mathrm{m2}}$ are used to avoid inconsistency. When voxel c has been processed by one thread, whether or not it needs to be further processed in the next iteration depends on the processing results of its adjacent voxels. The algorithm CUDA-FOE terminates when there is no connectivity update from any of the threads.

III. RESULTS

In this section, the running times of our GPU and CPU implementations of the FC algorithm are compared for image data of different sizes. The CPU version of FC is implemented in C++. The computer used is a DELL PRECISION T7400 with a quad-core 2.66 GHz Intel Xeon CPU. It runs Windows XP and has 2 GB of main memory. The GPU used is the NVIDIA Tesla C1060 with 240 processing cores and 4 GB device memory. CUDA SDK 2.3 is used in our GPU implementation. Three image data sets—small, medium, and large—are utilized to test the performance of the GPU and CPU implementations. Table I lists the image data set information and shows the performance of the GPU implementation versus the CPU implementation. A speed-up factor of 24.4x, 18.1x, and 10.3x, respectively, has been achieved, for the three data sets over the CPU implementation. It is noted that the segmentation results produced from both GPU and CPU implementations are identical. Here, the speed-up factor is defined as t_s/t_p, where t_s and t_p are the

TABLE I. Data set information and performance of the GPU implementation with respect to the optimal CPU implementation.

Dataset	Small	Medium	Large
Protocol	PD MRI	T1 MRI	CT torso
Scene domain	$256\times256\times46$	$256\times256\times124$	$512\times512\times459$
Voxel size(mm^3)	$0.98\times0.98\times3.0$	$0.94\times0.94\times1.5$	$0.68\times0.68\times1.5$
CPU time (s)	6.09	13.01	155.53
GPU time (s)	0.25	0.72	15.04
Speed-up	24.4	18.1	10.3

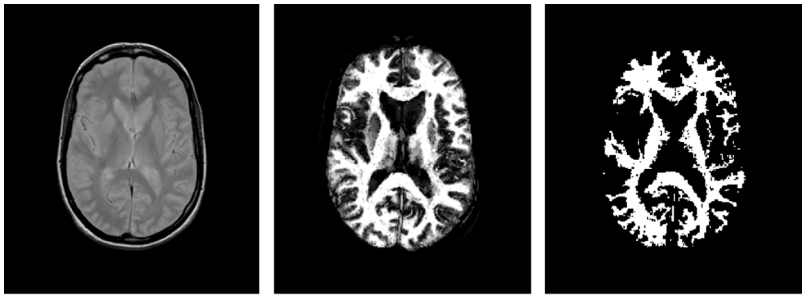

FIG. 3. A slice of the PD-weighted MRI scene from the small data set (a), the corresponding slices of the scenes depicting the connectedness values (b), and the final hard segmented white matter (c).

times taken for the sequential and parallel implementations, respectively. It seems that the larger the size of the testing data set, the less speed-up we achieved. This is mainly because for larger data sets, the affinity and tracking kernels require much more device global memory access, which has high latency.

Figure 3 shows the example of the small data set, which comes from MRI of the head of a clinically normal human subject. A fast spin-echo dual-echo protocol is used. Figure 3(a) shows one slice of the original PD-weighted scene, Figs. 3(b) and 3(c) show corresponding slices depicting connectedness values, and the final hard segmented white matter object.

Figure 4 shows the example of the medium data set, which is a T1-weighted MRI scene of the head of a clinically normal human subject. The spoiled gradient recalled (SPGR) acquisition was used. This data set were obtained from the web site of National Alliance for Medical Image Computing.[36] Figure 4(a) shows one slice of the original scene, and Figs. 4(b) and 4(c) show corresponding slices depicting the connectedness values, and the final hard segmented white matter object.

Figure 5 shows the example of the large data set, which is a CT scene of the torso. Figure 5(a) shows one slice of the original scene, and Figs. 5(b) and 5(c) show corresponding slices depicting the connectedness values, and the final hard segmented lung mask without pulmonary vessels.

IV. DISCUSSION AND CONCLUDING REMARKS

Recently, clinical radiological research and practice are becoming increasingly quantitative. Further, images continue to increase in size and volume. For quantitative radiology to become practical, it is crucial that image segmentation algorithms and their implementations are rapid and yield practical run time on very large data sets. This paper describes an example of practical and cost-effective solutions to the problem.

We developed a parallel algorithm of the widely used fuzzy connected image segmentation method on the NVIDIA GPUs, which are far more cost- and speed-effective than both COWs and multiprocessing systems. The parallel implementation achieves speed increases by factors ranging from 10.3x to 24.4x on Tesla C1060 GPU over an optimized CPU implementation for three image data sets with a wide range of sizes, and takes 0.25, 0.72, and 15.04 s,

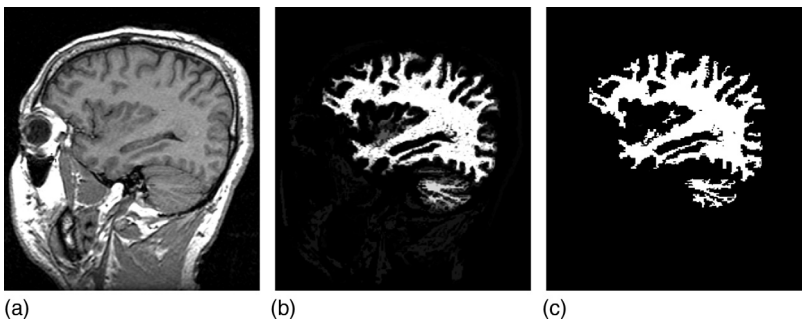

FIG. 4. A slice of the T1-weighted MRI scene from the medium data set (a), the corresponding slices of the scenes depicting the connectedness values (b), and the final hard segmented white matter (c).

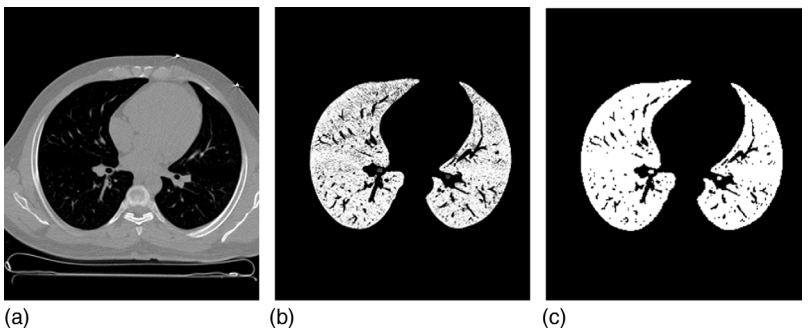

FIG. 5. A slice of the CT scene of the torso from the large data set (a), the corresponding slices of the scenes depicting the connectedness values (b), and the final hard object (c).

correspondingly, for the three image data sets. The performance of the parallel implementation could be further improved by taking advantage of fast GPU shared memory. A near-interactive speed of segmentation has been achieved, even for the large data set. For some specific applications, several free parameters (e.g., fuzzy affinity parameter, threshold value for the fuzzy object) in fuzzy connected image segmentation might be difficult to optimize.[8] The interactive speed of segmentation could give users immediate feedback on parameter settings; thus allowing them to fine-tune free parameters and produce more accurate segmentation results. Future work will also include the development of more advanced FC algorithms such as the scale-based relative FC (Ref. 5) and iterative relative FC (Ref. 31) on the GPU.

ACKNOWLEDGMENT

This research was supported by the Intramural Research Program of the National Cancer Institute, NIH.

[a]Electronic mail:zhugey@mail.nih.gov

[1]N. R. Pal and S. K. Pal, "A review on image segmentation techniques," Pattern Recogn. **26**, 1277–1294 (1993).
[2]D. Pham, C. Xu, and J. Prince, "Current methods in medical image segmentation," Annu. Rev. Biomed. Eng. **2**, 315–337 (2000).
[3]J. K. Udupa and S. Samarasekera, "Fuzzy connectedness and object definition: Theory, algorithms, and applications in image segmentation," Graphical Models Image Process. **58**, 246–261 (1996).
[4]P. K. Saha, J. K. Udupa, and D. Odhner, "Scale-based fuzzy connected image segmentation: Theory, algorithms, and validation," Comput. Vis. Image Underst. **77**, 145–174 (2000).
[5]J. K. Udupa, P. K. Saha, and R. A. Lotufo, "Relative Fuzzy connectedness and object definition: Theory, algorithms, and applications in image segmentation," IEEE Trans. Pattern Anal. Machine Intell. **24**(11), 1485–1500 (2002).
[6]G. T. Herman and B. M. Carvalho, "Multiseeded segmentation using fuzy connectedness," IEEE Trans. Pattern Anal. Machine Intell. **23**, 460–474 (2001).
[7]B. M. Carvalho, G. T. Herman, and T. Y. Kong, "Simultaneous fuzzy segmentation of multiple objects," Discrete Appl. Math. **151**, 55–77 (2005).
[8]Y. Zhuge, J. K. Udupa, and P. K. Saha, "Vectorial scale-based fuzzy connected image segmentation," Comput. Vis. Image Underst. **101**, 177–193 (2006).
[9]J. K. Udupa, L. Wei, S. Samarasekera, Y. Miki, M. A. Buchem, and R. I. Grossman, "Multiple sclerosis lesion quantification using fuzzy connectedness principles," IEEE Trans. Med. Imaging **16**(5), 598–609 (1997).
[10]J. Liu, J. K. Udupa, D. Odhner, J. M. McDonough, and R. Arens, "System for upper airway segmentation and measurement with MR imaging and fuzzy connectedness," Acad. Radiol. **10**(1), 13–24 (2003).
[11]E. Garduno, M. Wong-Barnum, N. Volkmann, and M. Ellisman, "Segmentation of electron tomographic data sets using fuzzy set theory principles," J. Struct. Biol. **162**, 368–379 (2008).
[12]Y. Zhou and J. Bai, "Multiple abdominal organ segmentation: An atlas-based fuzzy connectedness approach," IEEE Trans. Inf. Technol. Biomed. **11**, 348–352 (2007).
[13]T. Lei, J. K. Udupa, P. K. Saha, and D. Odhner, "Artery –vein separation via MRA—An Image Processing Approach," IEEE Trans. Med. Imaging **20**(8), 689–703 (2001).
[14]G. Moonis, J. Liu, J. K. Udupa, and D. Hackney, "Estimation of tumor volume using fuzzy connectedness segmentation of MRI," Am. J. Neuroradiol. **23**, 356–363 (2002).
[15]P. K. Saha, J. K. Udupa, E. F. Conant, D. P. Chakraborty, and D. Sullivan, "Breast tissue density quantification via digitized mammograms," IEEE Trans. Med. Imaging **20**(11), 792–803 (2001).
[16]G. Grevera, J. K. Udupa, D. Odhner, Y. Zhuge, A. Souza, S. Mishra, and T. Iwanaga, "CAVASS—A computer assisted visualization and analysis software system," J. Digit. Imaging, **20** (Suppl. 1), 101–118 (2007).
[17]B. M. Carvalho and G. T. Herman, "Parallel fuzzy segmentation of multiple objects," Int. J. Imaging Syst. Technol. **18**, 336–344 (2008).
[18]L. G. Nyul, A. X. Falcao, and J. K. Udupa, "Fuzzy-connected 3D image segmentation at interactive speeds," Graphical Models **64**(5), 259–281 (2003).
[19]GPGPU—general purpose computation using graphics hardware, http://www.gpgpu.org/.
[20]J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," Comput. Graph. Forum **26**, 80–113 (2007).
[21]S. S. Samant, J. Xia, P. Muyan-Ozcelik, and J. D. Owens, "High performance computing for deformable image registration: Towards a new paradigm in adaptive radiotherapy," Med. Phys. **35**(8), 3546–3553 (2008).
[22]B. Zhou, C. X. Yu, D. Z. Chen, and X. S. Hu, "GPU-accelerated Monte Carlo convolution/superposition implementation for dose calculation," Med. Phys. **37**(11), 5593–5603 (2010).
[23]J. E. Cates, A. E. Lefohn, and R. T. Whitaker, "GIST: an interactive, GPU-based level set segmentation tool for 3D medical images," Med. Image Anal. **8**, 217–231 (2004).
[24]M. Roberts, J. Packer, M. C. Sousa, and J. R. Mitchell, "A work-efficient GPU algorithm for level set segmentation," *Proceedings of the ACM SIGGRAPH/Euro-graphics Conference on High Performance Graphics* (2010).
[25]L. Grady, T. Schiwietz, S. Aharon, and R. Westermann, "Random walks for interactive organ segmentation in two and three dimensions: Implementation and validation," *Proc. MICCAI*, **2**, 773–780 (2005).
[26]V. Vineet and P. J. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops, pp. 1–8 (2008).
[27]K. C. Ciesielski and J. K. Udupa, "A framework for comparing different image segmentation methods and its use in studying equivalences between level set and fuzzy connectedness frameworks," Comput. Vis. Image Underst. **115**, 721–734 (2011).
[28]K. C. Ciesielski, J. K. Udupa, A. X. Falcao, and P. A. V. Miranda, "Comparison of fuzzy connectedness and graph cut segmentation algorithms," Proc. SPIE Med. Imaging, **7962**, 17 (2011).
[29]R. Audigier and R. A. Lotufo, "Duality between the watershed by image foresting transform and the fuzzy connectedness segmentation approaches," *Proceedings 19th Brazilian symposium on computer graph and image processing (SIBGRAPI'06)*, pp. 53–66 (2006).
[30]Y. Zhuge, Y. Cao, J. K. Udupa, and R. W. Miller, "GPU accelerated fuzzy connected image segmentation by using CUDA," *Proceedings of IEEE Engineering in Medical and Biology Society*, pp. 6341–6344 (2009).
[31]K. C. Ciesielski and J. K. Udupa, "Region-based segmentation: fuzzy connectedness, graph cut and other related algorithms," in *Biomedical Image Processing*, edited by Thomas M. Deserno, (Springer Verlag), 251–278 (2011).
[32]K. C. Ciesielski and J. K. Udupa, "Affinity functions in fuzzy connectedness based image segmentation II: Defining and recognizing truly novel affinities," Comput. Vis. Image Underst., **114**, 155–166 (2010).
[33]NVIDIA Corporation, NVIDIA CUDA programming guide 2.3, http://developer.nvidia.com/cuda, July, 2009.
[34]A. S. Nepomniaschaya and M. A. Dvoskina, "A simple implementation of dijkstra's shortest path algorithm on associative parallel processors," Fund. Inform. **43** (1–4), 227–243 (2000).
[35]P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *High Performance Computing 2007*, Vol. **4873**, Lecture Notes in Computer Science (Springer, New York, 2007), pp. 197–208.
[36]http://www.na-mic.org