제 116 회 석사학위논문
지도교수 손 봉 수

관심 컨투어의 추출을 위한 혼합 가속화 방법에 대한 연구

Hybrid Acceleration for Interest-based Contour Component Extraction

중앙대학교 대학원

컴퓨터공학과 컴퓨터공학 전공

진 량 보

2012년 2월

관심 컨투어의 추출을 위한 혼합 가속화 방법에 대한 연구

Hybrid Acceleration for Interest-based Contour Component Extraction

이 논문을 석사학위 논문으로 제출함

2012 년 2 월

중앙대학교 대학원

컴퓨터공학과 컴퓨터공학 전공

진 량 보

진량보의 석사학위 논문으로 인정함

심 사 위 원 장 _____ 인

심 사 위 원 _____ 인

심 사 위 원 _____ 인

중앙대학교 대학원

2012 년 2 월

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1. Introduction

Medical imaging, which is used to create images of the human body for clinical purposes, plays an important role in medicine. When a set of images is captured, 3-dimensional volume data can be made. Many applications were made to visualize the volume either by *volume rendering* [6] or *marching cubes* [13] algorithms. These application were further developed to emphasis the visualization of interest-based areas in the volume. Typical designs were made by C.L. Bajaj. et al. for designing the *contour spectrum* [1], which is used for finding important isocontours. On the other hand, G. Kindlmann et al.[11] designed proper transfer function to provide better view of direct volume rendering, so that medical applications are made possible. In recent years, these methods could be carried out with GPU acceleration [12], which is lower in cost but far more efficient in computing and rendering.

However, contours extracted using the marching cubes method won't deliver a clean image of a 3D surface. The output usually contains some noisy contours that might block the region of interest. To satisfy the need of analyzing the volume and extracting clean 3-dimensional mesh, *contour tree* [19] is introduced and heavily used as the abstraction of the 3D images. And then surface can be traversed starting from a group of seed cells by propagating through neighboring cells using adjacency and intersection information.

Although both structured and unstructured meshes can be analyzed using the contour tree structure, noisy data such as MRI and CT scans can cause the algorithm to produce large numbers of seed cells, which would cost longer preprocessing time. To solve this problem, H. Carr et al. [4] proposed a tree simplification method to reduce branches by local geometric measures, while V. Pascucci et al. [16] introduced a branch decomposition method with 3-dimensional presentation of contour tree. The branch decomposition algorithm aims to represent every arc appears in exactly one branch. Following this rule, except the root branch, all branches in the tree connects one leaf to an interior node of another branch. Then apply valid simplification function to peel off branches. This approach would result in a hierarchical representation of the contour tree. Later, J. Zhou et al.[20] improved this approach with different geometric measures and result in improved more reasonable simplified layout.

The central idea of the original contour propagation method is that, when found neighboring cells intersect with the initial cell (seed cell), put them into a queue so that more neighboring cells can be reached with the same queue. A difficult portion in this algorithm lies in locating and triangulate the neighboring cells. And for irregular meshes, this could be efficient to trace by performing a breadth-first search in the graph of cell adjacencies.

There are two major contributions in this thesis. Firstly, for improving the speed of contour propagation, we applied our parallel algorithm to take the advantage of modern multi-core processor. This stage is proceeded in CPU mainly because of the queue-based contour propagation algorithm. And we generate a compact array, which contains the size of triangles, for parallel triangulation during the process. Secondly, with the help of the precomputed compact array, we proposed a GPU-accelerated method for triangulate the active cells and render them

immediately after triangulation. Advantages of this method will be described in Chapter 4.

The basic idea for computing contour propagation in parallel is inspired by the task stealing method[14], which provides load balancing while compromise data locality. The starvation of a thread can be solved by stealing a seed cell from the tail of another instance, therefore, each thread could preserve locality of its seed cell queue.

With the combination of seed cells collected from the above parallel computation, all tetrahedral cells are triangulated in GPU. The algorithm improve the efficiency of computing by avoiding visiting cells without intersection with given isovalue and directly rendering triangulated mesh after GPU triangulate execution.

The overall workflow that how our program process the 3-dimensional volume data is given in Figure 1. The computation of contour tree is considered to be the preprocessing stage. And in the interactive stage, regions of interest are selected using an interface to extract the contour using the proposed hybrid accelerated method. This highly depends on a simplification process of contour tree to visualize noisy biomedical data.

The rest of the thesis is organized as follows. The method we used for finding Region of Interest, most of which are previous works, are described in Chapter 2. And then a hybrid accelerated method for propagating the contour and triangulating active cells is introduced in Chapter 3 and Chapter 4 respectively. Chapter 5 describes the experiment environment and presents the results. Finally, a summary of the thesis and the conclusions are drawn in Chapter 6 .

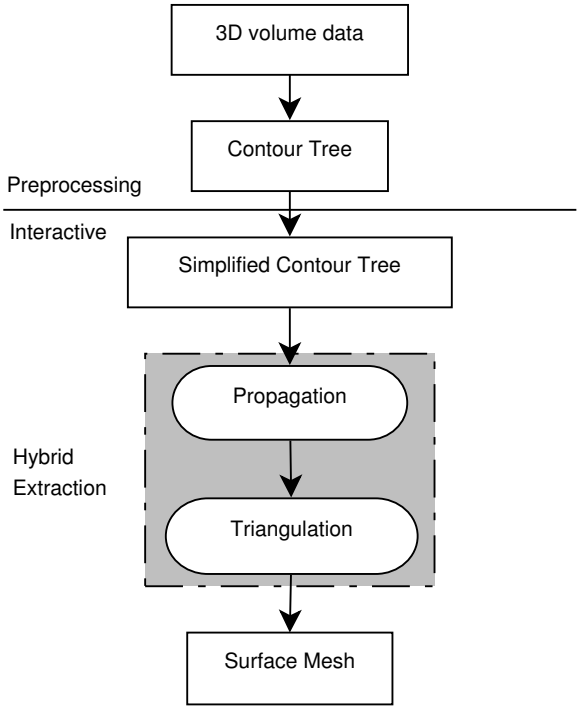


Figure 1: Overall Workflow

Chapter 2. ROI Computation

This chapter describes the region of interest computation method, most of which focused on the preprocessing of 3-dimensional volume data, which aims to find out the topological structure. In our approach, contour tree is used as the abstraction of the volume data. Although this won't be good when the input data is noisy, fast contour extraction method depends on the generated seed set during preprocessing. And when applied simplification methods, it won't be hard to find significant contour structures in the original volume.

2.1 Background Research

The concept of an ROI, which is a selected subset of samples in a image, is commonly used in medical imaging. Typical application covers the dimension from 2D image to volumetric data and time-varying volumes and focus on segmenting boundaries in an image or a volume.

2.2 Contour Tree

H. Carr proposed the concept of *flexible isosurface* [5] and the simplification method to get clean output of isosurface mesh. There are several terms defined in this work,

such as *path seeds* and *local geometric measures* etc.

The local geometric measures can be used to validate individual contours when simplifying the contour tree to a hierarchical structure. The basic measure is to use *topological persistence*, which is the difference in function value on an edge that connects a pair of critical points. Several other measures like *surface area* and *integral volume* are used for further exploration of the volume.

Contour tree for 3D images is introduced by M. Van Kreveld et al.[19] to improve the speed of isocontouring. The contour tree is further developed by H. Carr [3] to help the interactive exploration of multi-dimensional imaging data. It is widely used in various field of study, including data compression, contour matching[17], GIS etc. And in various data dimension from 2D images, 3D volume data and 4D time-varying volume.

There are various implementations of computing Contour Tree from 2D/3D dataset. And among those implementations, H. Carr's method is well-known and widely accepted as the general method, which takes $O(N \log N)$ in run-time for Contour Tree construction.

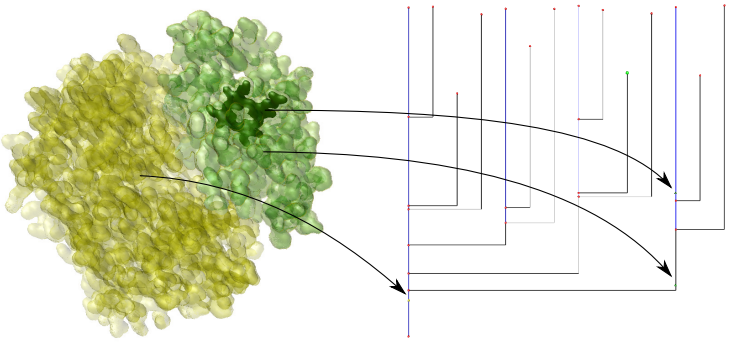


Figure 2.2: Contour tree mapped to 3D volume data

2.2.1 Contour Tree Construction

To generate the contour tree information, we follow the conventional method to create it in $O(n \log n + t\alpha(t))$. We sort the intensity values of all voxels in the volume, sweep from high to low isovalues to construct join tree using union-find structure[18] to determine connected components.

The algorithm for computing contour tree described by H. Carr et al.[3] consist of the following four steps.

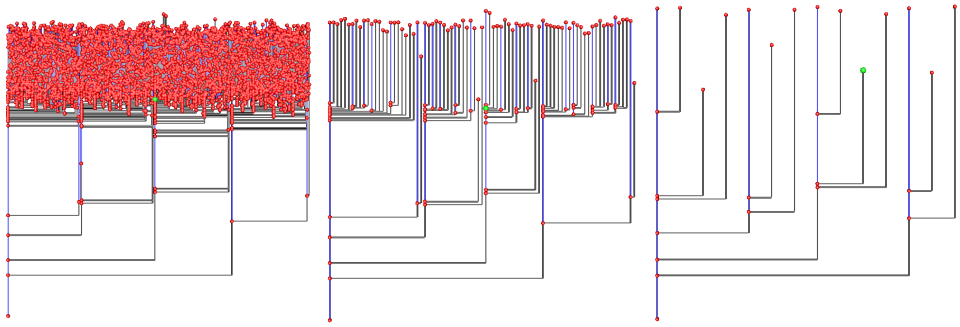
1. *Sort all* n *vertices of the mesh by their intensity values.*
2. *Perform a sweep of the* n *vertices from lowest values to highest ones to construct* join tree.
3. *Perform another sweep from a reverse direction to construct* split tree.
4. *Merge the join tree and split tree and remove nodes that keep no topological changes.*

To track the individual contour components in the volume data, store edge information as the seed during the construction of join tree. Then transfer it to the contour tree during the merge step of the algorithm. In the meanwhile, we generate one and only one seed for each contour depend on the contour tree information. This provides the information to manipulate and annotate individual contours interactively.

In our implementation, the generated seed is stored with contour tree data file in volume data preprocessing stage. The information can be reused to generate simplified contour tree interactively as well.

2.2.2 Branch Decomposition & Contour Tree Simplification

V. Pascucci et al.[16] composed a multi-resolution data structure for representing the contour tree in a hierarchical layout and an algorithm to construct it.



(a) coarse simplified 1 (b) coarse simplified 2 (c) fine simplified

Figure 2.3: Interactive contour tree simplification

We implemented the contour tree simplification according to the persistence value of each branch, which is the length of the branch. Each saddle-extremum pair, whose persistence is below a given threshold is considered to be insignificant. Therefore, we remove the arc that is consider to be unimportant from the contour tree to discard the corresponding noisy structures. Besides, we present an interface that can provide threshold interactively to visualize simplified contour tree.

2.3 Contour Property Computation

C.L. Bajaj et al. [1] introduced *contour spectrum*, which provides global geometric properties like surface area, volume and gradient integral of the contour. This work enables the identification of important isovalues for guiding exploratory visualization through a simple interface. V. Pascucci et al. [15] propagate topological indexes along branches of the contour tree. H. Carr et al. [4] later defined local

geometric measures for individual contours, such as surface area and contained volume.

Surface Area Computation Surface area implies the exposed area of a solid object. For instance, surface-area-to-volume ratio(SA:V) of a sphere is small. In the contrary, the SA:V ratio of many body parts, say brain, are very large due to infoldings, allowing higher rates of metabolism.

We use the additivity property of the surface area add up non-overlap triangles of the surface.

$$A(S) = A(S_1) + \cdots + A(S_r).$$

Gradient Computation The gradient of a scalar function $f(x_1, x_2, x_3, \ldots, x_n)$ is denoted ∇f. The notation $\text{grad}(f)$ is also used for the gradient.

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right).$$

In 3-dimensional rectangular coordinates, this can be expanded to

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

Therefore, for each triangle i, we compute the summation of normal vector length and divide it by the area.

$$g_i = \frac{|\vec{n}_1| + |\vec{n}_2| + |\vec{n}_3|}{3 \times area}$$

And the gradient of the contour component would be the average value of those triangles.

$$G = \frac{\sum_{i=1}^{n} g_i \times a_i}{\sum_{i=1}^{n} a_i}$$

Volume Computation Complex shapes like brain surface can be calculated by *integral calculus* in the condition that a formula exists for the shape's boundary. Since the procedure of building contour tree sweep every vertices in the volume and results in contour tree edges which represent for monotone structures. The summation of all the cells on the monotone structures, which are adjacent to each other on the same direction, is the volume corresponding to the contour tree edge. This is the information commonly used for estimating whether a contour is the region of interest.

Chapter 3. Contour Propagation (Multi-core CPU)

In this chapter, we describe a novel algorithm to propagate an individual contour from a single seed. The seed is generated during the preprocessing stage while constructing the contour tree. It is difficult to propagate a large contour using a parallel method because it's hard to predict the direction and position the contour. Therefore, we describe our algorithm in this chapter as the main contribution of the thesis.

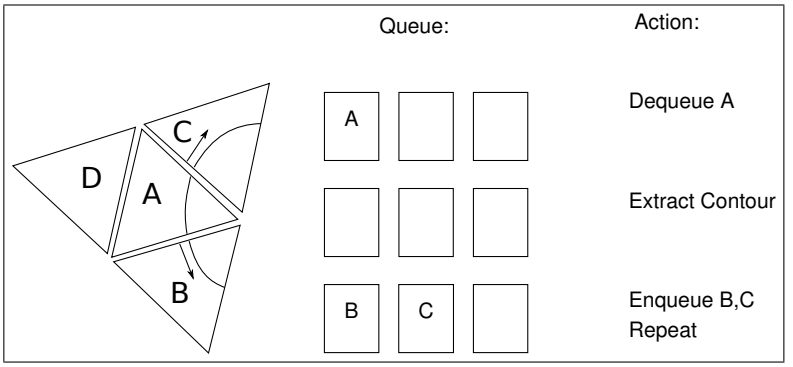


Figure 3.4: Illustration of contour propagation

3.1 Background Research

In this thesis, we focus on extract the interactively selected large contour component while taking advantage of modern hardware. Conventional sequential extraction method won't meet our need for real-time interest-base contour component visualization.

The commonly used method for propagating an individual contour is to compute the intersected neighboring cell by comparing vertex values with the given isovalue. Then push all the intersected neighboring cells into a queue so that those cell can be triangulated in later steps. For the case used in this thesis, 6 tetrahedrons are divided from a single cube. Therefore, the intersected tetrahedron might be a tetrahedron in another cube.

The problem we met to complete the contour propagation in a parallel way is how to implement the push and pop operations efficient, while propagating from a single seed cell to all directions. A naive method that could be think about is to use a shared queue between all threads while all operations on the queue should be protected by mutex lock. This could be much more efficient than the sequential method due to the reduction of the time to compute intersected neighboring cells. Because these operations can be done in parallel. However, the time to push and pop in the shared queue is not reduced at all. And this could be a major obstacle to improve the contour propagation.

3.2 Concurrent Propagation

To avoid long time waiting for execution to access shared queue, we designed an algorithm to maintain a shared queue and use local queue in each thread, while use a shared queue to avoid the situation of empty queue starvation.

To prevent duplication when visiting the volume vertices, we maintained a shared set between all the threads. Since we just store whether the vertex is visited in the set, the type of is simply a bitset. Although it is shared between all the threads, it is rarely visited during the propagate progress. Additionally, the bitset is initialize as 0 and set to 1 when visiting the cell. Therefore, it's safe to maintain it as a lock free set.

Algorithm 1: *propagate* Function

Input: Isovalue, Initial Seed Cell, $ThreadID$, Total Number of Threads, Initialized Unvisited Bitset
Output: $ActiveCellArray$, Parallel Prefix Sum($Scan$)

```
1  Create LocalQueue;
2  while True do
3      Cell ← dequeue();
4      Get Vertex and FunctionValue of Cell;
5      Compute Index by compare(FunctionValue, Isovalue);
6      ActiveCellArray[ThreadID] ← Cell and Index;
7      foreach intersected adjacent tetrahedron do
8          AdjacentCell ← GetNeighborCell(Cell);
9          if AdjacentCell exists & not visited then
10             Visist(AdjacentCell);
11             enqueue(AdjacentCell);
12         end
13     end
14 end
```

At the starting point of propagation, one of the threads get a seed cell index. To find intersection points, sampling current cell vertices and function value and comparing the function value to given isovalue to get intersection position. In our case of propagating a tetrahedral mesh, four sampled values should compare to the isovalue. Then store cell index and intersection information into each thread specific active cell array. This array is later needed as a collection of active cells and its information to generate parallel prefix sum. Then sample neighboring cells

with intersection and enqueue it.

3.2.1 Usage of Local Queue

The trick to use local queue in each thread for contour propagation, which guarantees no seed loss, mainly focused on how to get a seed. To describe Algorithm 2 the following paragraph gives out a summary of the idea:

To get a cell each time, local queue is visited first. And if there is no cell in local queue, the thread will announce that it lack of seed cell for further propagation, so that other threads might aware this and push a cell into the shared queue. Then go on to check out whether all the other local queues are empty. Now that all the local queues and shared queue are empty, it means that all the propagation progress is finished and the instance of current thread should be terminated. But if the local queue in one of the other threads is not empty, the current thread will go on to the next loop to find out whether a seed cell is pushed into the shared queue.

On Line 6 of Algorithm 2, we check the size of shared queue and then dequeue it immediately after the check in the same function with lock protection. Otherwise, it can't guarantee there is element while dequeue it from shared queue, because the instance of shared queue might be different between the time when check the size of the queue and dequeue function is called.

Consequently, when propagating with a seed cell, the neighboring cells will be pushed into local queue if all other threads didn't make the lack of seed announcement. Otherwise, the new generated seed should be pushed into the shared queue to meet the needs of local queue(s) of other threads.

Algorithm 2: *dequeue* Function

```
1  if LocalQueue is not empty then
2  |   Cell ← Pop(LocalQueue front element);
3  else
4  |   while True do
5  |   |   LackOfSeedFlags[ThreadID]← True;
6  |   |   if Cell ←Dequeue(SharedQueue) failed then
7  |   |   |   if all LackOfSeedFlags is False then
8  |   |   |   |   TerminateThread(ThreadID);
9  |   |   |   else
10 |   |   |   |   continue;
11 |   |   |   end
12 |   |   else
13 |   |   |   LackOfSeedFlags[ThreadID]← False;
14 |   |   |   break;
15 |   |   end
16 |   end
17 end
```

Algorithm 3: *enqueue* Function

```
1  if all LackOfSeedFlags is False then
2  |   LocalQueue ← Push(AdjacentCell);
3  else
4  |   SharedQueue ← Push(AdjacentCell);
5  end
```

3.2.2 Post-processing

For each active tetrahedral cell in active cell array, there might be one or two corresponding triangles, which depends on the intersection position on the tetrahedron. To finish this step in an efficient way, a simple CUDA kernel was made to generate the array for generating mesh in parallel pipeline while avoiding conflicts between CUDA threads.

By comparing given isovalue with intensity values of neighboring cell to get the intersected neighbors next to it, we can get the parallel prefix sum [9] of each

Sequential	Parallel
for *j* from 1 to n **do** scan[*j*] = scan[*j*-1] + f(in[*j*-1]);	**foreach** *j* in parallel **do** scan[*j*] = f(in[*j*]);

Table 3.1: Comparison of sequential and parallel *scan* generation

active voxel. [1] This can be used for predicting the size of output triangles in CUDA memory.

[1] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Chapter 4. Active Cell Triangulation (Many-core GPU)

Due to the parallel architecture on modern graphics cards, a mount of graphics related as well as general purpose applications are developed on top of GPU. Those applications might be implemented in shader languages like GLSL or general purpose computing languages like OpenCL. And among those implementations, NVIDIA's CUDA architecture is widely used due to the highly optimized CUDA libraries and regularly updated SDK. Even many of the modern supercomputers consist of thousands of NVIDIA's GPUs.

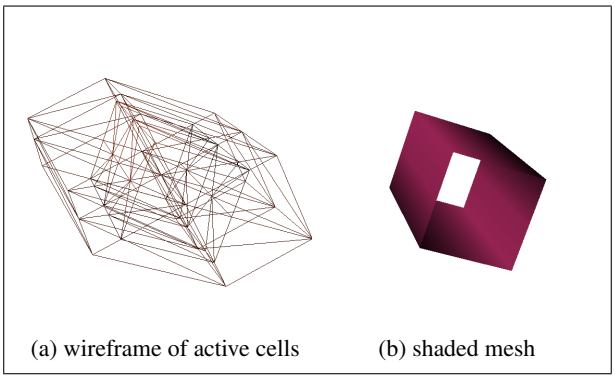


Figure 4.5: Cell triangulation (only tetrahedral cells with intersection is triangulated in GPU)

4.1 Background Research

For triangulate active cells extracted from a initial seed cell using geometric propagation method, GPU implementation is a decent choice to consider. C. Dyken et al.[7] composed a method to enable MC algorithm running on GPU using shader programming while taking the advantages of a data packing structure called *HistoPyramids*.

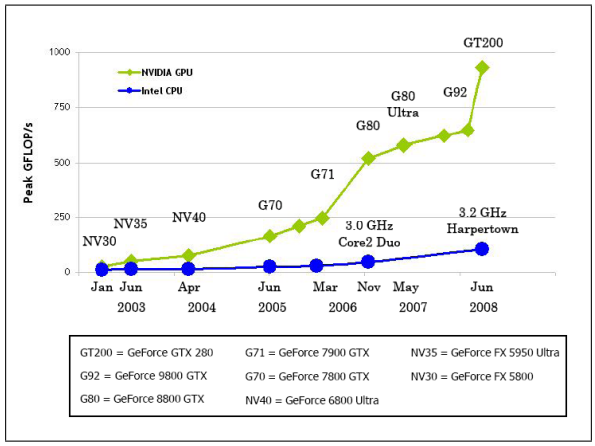


Figure 4.6: Difference between CPU and GPU in parallel computing

Vertex shader based active cell triangulation [8] focus on the technique to pack the active cell data to compact them as small as possible to improve the transfer speed. And even intensive values can be transferred to GPU during the rendering process. It's good to render dataset that is large in volume, due to the algorithm don't require a initialize step to put all volume to GPU as texture.

All of these implementation got a common problem on copying GPU precomputed mesh into main memory to prevent duplicated computations. That is to said it will triangulate active cells even when operations like rotation was made. It cost much unnecessary GPU computation when using shader programs for triangulation.

In general, shader-based triangulation method is much easier to implement, but far less efficient comparing to the general-purpose GPU programming methods due to the difference of graphics programming implementation. Therefore, it's the limitation of the shader programming method for extracting isosurface[8].

The following is a comparison between the three widely used GPU programming APIs:

- **OpenCL**: Khronos Group's OpenCL cross-platform and implemented on both Nvidia and AMD's GPUs. It can be easily ported to multi-core exections. However, it is less developed comparing to CUDA.
- **CUDA**: Nvidia's CUDA is currently most highly developed GPU programming architecture. It features in various GPU memory management and graphics related interoperability. However, it is only available on Nvidia's graphics cards.
- **DirectCompute**: DirectX compatibility, but limited to Microsoft Windows 7 and Vista.

Advantage of GPGPU programming : save proceeded data into memory and use synchronize method to guarantee to correctness of computing and improve the speed further.

Faster large object rendering method for preventing overhead of calling data input functions.

4.1.1 Parallel Contour Extraction Methods

C.L. Bajaj et al.[2] designed out-of-core algorithm aim to extract huge dataset using a parallel method. Depending on the static analysis information precomputed,

the algorithm focused on loading balance and minimize secondary memory access. A volumetric dataset is divided into the atomic processing elements called *blocklet*. Therefore, only cells intersected with a range of isovalue would be loaded into main memory. This approach efficiently reduced the data loading to avoid visiting blocklets that don't contain active cells. Similar work was done to the GPU processing method. Goetz et al.[8] developed the method to carefully compact active cell data as texture and extract the surface using vertex shader. The algorithm aim to reduce the data transaction by concatenate the vertex data with a bitwise operation, so that small portions in large volume data can be visualized with GPU acceleration. However, the performance of this work mainly depend on the speed of data transaction to the graphics card.

4.1.2 Parallelism in CUDA Infrastructure

Note that an implementation of the Marching Cubes method [10] exist in recent version of CUDA SDK. This program takes the advantage of the parallel structure of CUDA to extract all of the cells in the given volume data. Although it's fast enough to extract large volume data in real-time, but obviously it won't be possible for this program to process datasets as huge as Giga Bytes. Consequently, active cells that are needed to be triangulated should be preprocessed before loading the whole volume data into CUDA memory. A detail solution is given in Chapter 4.

Figure 4.7 illustrates the relationship between host and graphics device. The computing units on each graphics card are divided into several grids, and each grid consists of a number of blocks, where many threads could be run in parallel. And the execution is enabled to assign the work to each block, therefore, it's efficient to use CUDA for the computation of large amount of simple works.

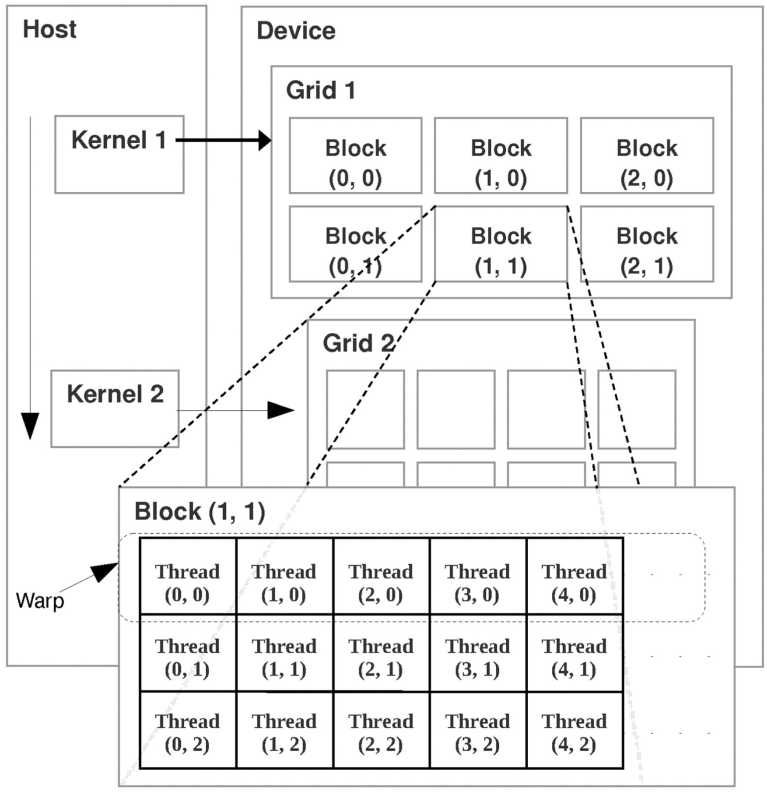


Figure 4.7: CUDA Execution Model

4.2 Active Cell Triangulation

Right after the active cell array is computed, a GPU array can be allocated same size of the active cell array, where consist of the continuous indices of tetrahedron intersected with given isovalue.

Note that in order to triangulate the volume data that is larger than GPU memory, volume data can be allocated and compacted after active cell array computation instead of loading the whole volume. However, this would slow down the triangulation time and might be slower than CPU depending on the size of the chosen contour.

Taking the advantage of CUDA's interoperability with OpenGL[2], vertex buffer object(VBO) is mapped into the address space of CUDA before launching triangulation computation kernel. And right after the CUDA triangulation, buffer object is unmapped from device memory address. [3]

There are three tables that can be loaded as texture[4]. One is the triangle table that maps same tetrahedron vertex index to a list of up to 2 triangles which are built from interpolated edge vertices. And in the case of triangulation on a single cube, there would be up to 5 triangles. The reason cubic data in the volume are divided into tetrahedron is mainly because of the ambiguity problem when building the contour tree. Dividing the cube avoids the ambiguity problem and guarantees the topology equivalent property between the contour tree and volume data. Two other tables are an edge table which is used for mapping vertices to edges that are intersected and a cube dividing table which is used for storing tetrahedron connectivity information in a cube.

The most difficult problem to the triangulation tetrahedron cell is how to organize vertex index and and put interpolated information into global memory without conflict. To solve this problem, cell intersection on edges are estimated after combining active cell in CPU.

With compact array (parallel prefix sum), we perform linear interpolation on each intersect edge of all active cells.

[2]In fact, Direct3D interoperability is also available. In our implementation, to enable the program both in Win32 and Linux, OpenGL functions are used.

[3]Note that a buffer object must be registered to CUDA before it can be mapped. This is done with `cudaGLRegisterBufferObject` function.

[4]However, for tables that are small in size, it's good to put it directed into device memory for high speed of access during execution.

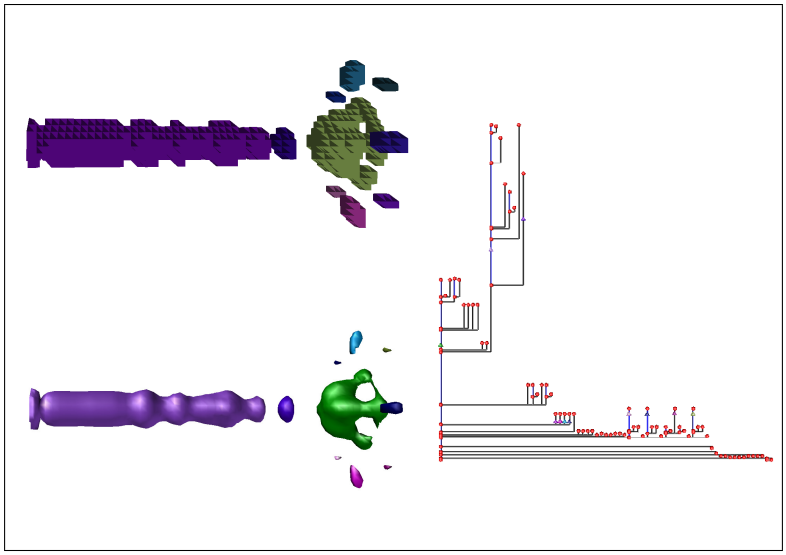


Figure 4.8: Only cell blocks with intersection are copied to GPU for triangulation, this improved the efficiency by avoid visiting empty cells.

For Surface smoothing, surface normal is computed during the triangulation process. In our implementation, we use Gouraud shading method for balance the fast processing and rendering quality. The computation of Gouraud shading produces continuous shading of surfaces by estimating surface normal of each vertex in the polygonal 3D model. The normal on each vertex are set to be the average value of all neighboring vertices.

4.3 VBO-based Rendering

There are several reasons to apply VBO rendering to the implementation: First, it is a time-consuming task to cost memory data from GPU to main memory. Second, `glVertex*` has the potential to be slower, because there is more call overhead than `glDrawElements`. Further more, it's duplicate data transfer between main memory and GPU.

Due to the usage of parallel prefix sum in our method, the GPU memory to put all triangles are allocated as vertex buffer object(commonly known as VBO) immediately after the current propagation stage. This section of global memory in GPU is then filled with triangulated mesh data during GPU-based triangulation. VBO is used to reduce of avoid large amount of function calling while rendering triangulated mesh and reduce data transfer from between GPU and CPU.

Chapter 5. Results

In our tests, we use a Linux system (Debian 6.0 Distribution) equipped with a hexa-core processor (AMD Phenom II X6 1055T) running at 3.30 GHz, 8GB RAM, as well as Nvidia GeForce GTX 460. The graphics card is enabled with CUDA compute capability 2.1. The graphics card specific CUDA compute capability table are available on the Nvidia's official site[5]. Tested datasets are vary from molecule data to large scan medical image volume data, many of which were provided on *http://www.volvis.org*.

In this thesis, a comparison on performance is made between conventional approach and the composed new method. The results largely depend on the hard limitations such as bandwidth between CPU and GPU, the number of cores exists in CPU and GPU. However, on most modern computers equipped with Nvidia's GPU, relevant fast contour visualization is achieved as is shown in Figure 5.9.

5.1 Propagation Performance

Here we make a comparison between our proposed results with original sequential method. It's not hard to find out the speed improvement from Table 5.2. The performance highly depends on the usage of threads, but too many thread usage

[5]http://developer.nvidia.com/cuda-gpus

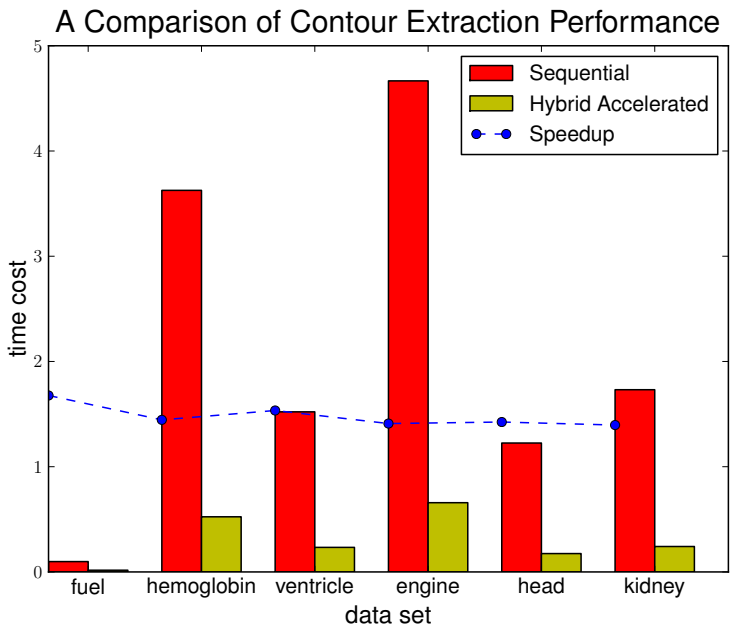
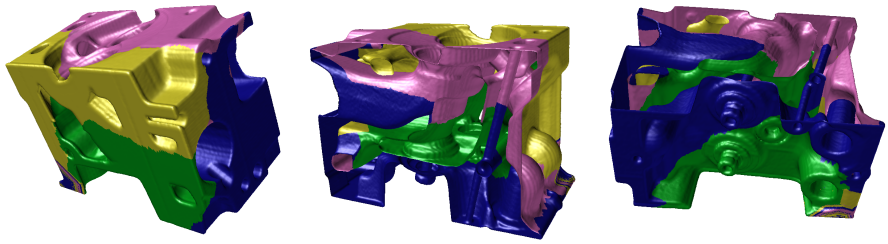


Figure 5.9: A performance comparison between conventional sequential program and the composed hybrid accelerated method. (A stable speed improvement up to 6∼7 times is shown in blue curve)

will lead to overhead global queue usage. Consequently, the number of threads is usually configured as many as the number of cores in the CPU to get best performance on the computer.

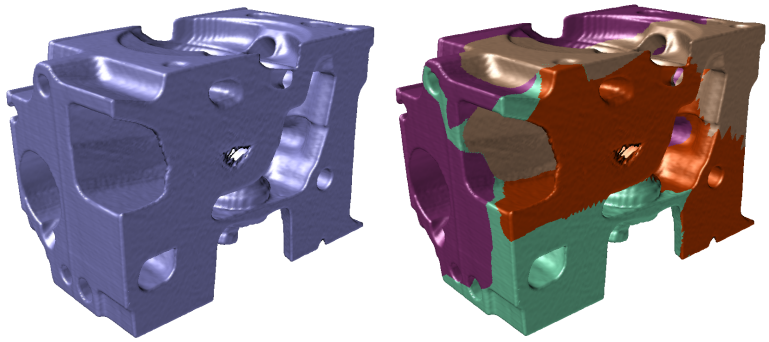
Results shown in Figure 5.10 indicate that the proposed algorithm is efficient in assigning work to different threads. Most of the active cells are continuous because they are extracted using the same seed cell. The active cells extracted by all threads are almost equal except for some rare cases when conflict among threads take place. The conflict might take place on the beginning of the propagation due to the starvation among the number of threads.

From Figure 5.12 we can see that when using a dual-core processor, best performance can be obtained when using 3∼5 threads. For the propagation of a large



(a) front (b) back (c) side

Figure 5.10: A contour component extracted from engine image data, where color shows contribution of different processors.



(a) color-tagged (b) shaded

Figure 5.11: Comparison between (5.11a) shaded contour component and (5.11b) color-tagged model, where slight thread conflict take place during initial propagation.

contour, the speedup is much more efficient using threads more than the number of processors. This mainly because when conflict take place between two threads, the existence of other threads can continue to propagate. Therefore, the number of threads used in 1.5 times of that of cores to estimate best performance based on the available hardware. The figure also shows that starting from using a single thread, with the increase of thread use, the time for propagating dropped gradually due to the multi-tasking function on different computing units. The drop after obtaining

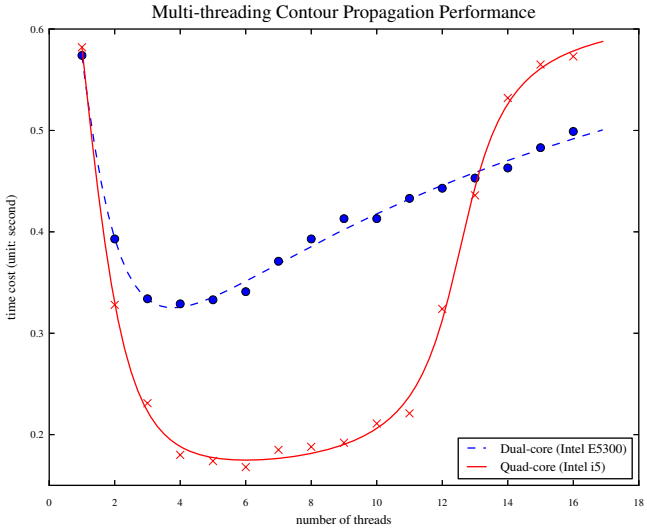


Figure 5.12: Multi-threading contour propagation performance tested on hemoglobin data. (y axis shows time to obtain 1,237,574 tetrahedral cells, while lines show fitting function: $f(x) = \frac{\alpha}{x^{\beta}+\beta x+\gamma} + \delta arctan(\varepsilon x + \zeta) + \eta$

best performance is contributed by overhead host function calling and too much threads compete to get seeds from shared queue. But for contours small in size, it's efficient to use threads same as the number of processors.

5.2 Triangulation Performance

GPU performance could be slightly affected by several factors, such as the distribution of the GPU processing load, the bandwidth between GPU and CPU, as well as memory size in graphics cards.

load balancing The distribution of the GPU processing load is assigned by setting the number of threads used in each block, which is a multiple of 2. In our experiment, it's proper to set the value in range from 32 to 128. That is to say that each block processes 32$\sim$128 tetrahedral cells at a time, while each cell results

in 1$\sim$2 triangles. Taking advantage of the *vertex buffer object*(VBO), all these triangles in the device addresses can be mapped into rendering pipeline by binding buffer IDs and enable the vertex buffer in display function.

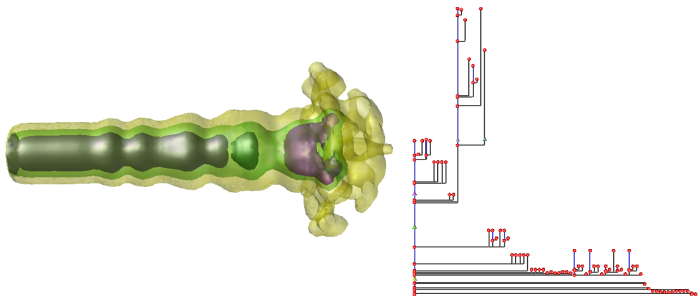
The amount of work that assigned to each thread are slightly different because the number of triangles inside a tetrahedron can be 1 or 2 depending on the intersection case. The time cost on execution depends on the slowest thread. Therefore, the time cost in most case is the time consumed on extracting two triangles intersected in a single cell.

bandwidth From Table 5.2, we can see that bandwidth is an important factor affecting the GPU computing performance. Active cells collected using multitasking should be copied into GPU memory, which is allocated and released every time. Therefore, data that need to be loaded should be small in size and in less frequency. In our experiment, computing cell index on intersection is much more efficient than loading precomputed array. Because the computation on GPU is executed on parallel pipeline, while loading arrays to global memory is run in linear time complexity $O(n)$.

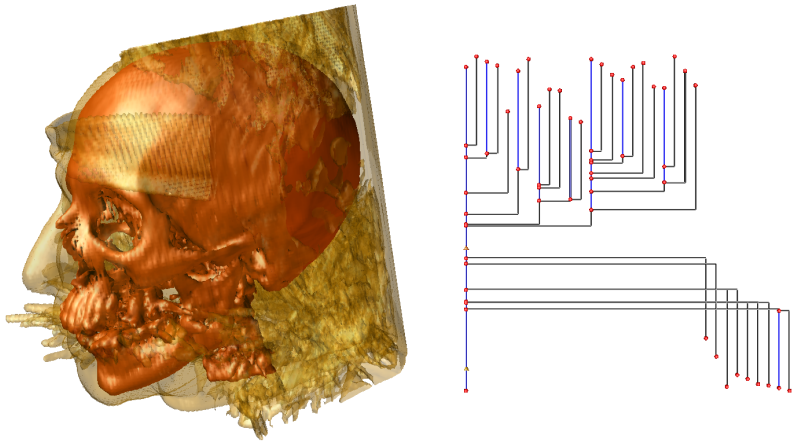
miscellaneous In practice, CUDA SDK version is another factor that affects the final triangulation performance. Newer versioned toolkit might be much more optimized to achieve better performance.

data set	dimension	triangles	active cells	previous	propagation		triangulation		proposed	speedup
					original	proposed	original	proposed		
fuel	64x64x64	33618	25613	0.0984	0.016	0.009	0.056	0.007	0.0165	x5.96
visible human	128x128x128	471107	357986	1.0545	0.220	0.053	0.829	0.086	0.1517	x6.95
head	128x128x128	573840	436870	1.2250	0.264	0.058	0.955	0.103	0.1745	x7.02
kidney	128x128x128	769305	586956	1.7316	0.362	0.086	1.364	0.138	0.2417	x7.16
hemoglobin	128x128x128	1619392	1235133	3.626	0.763	0.194	2.856	0.294	0.524	x6.9
ventricle	256x256x128	643171	491264	1.5214	0.312	0.069	1.172	0.118	0.2334	x6.52
engine	256x256x128	2060552	1561463	4.6664	0.986	0.212	3.639	0.369	0.6580	x7.09

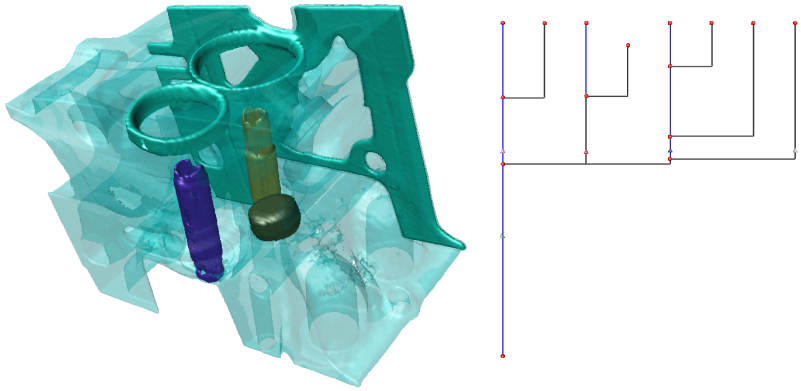
Table 5.2: Performance Analyzing Table (In order to prove the efficiency of the proposed algorithm, we selected large contours in the dataset for testing the performance.)



(a) fuel

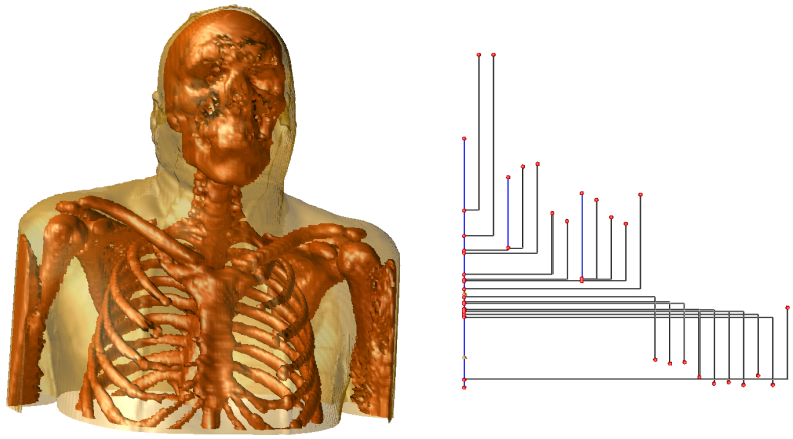


(b) head

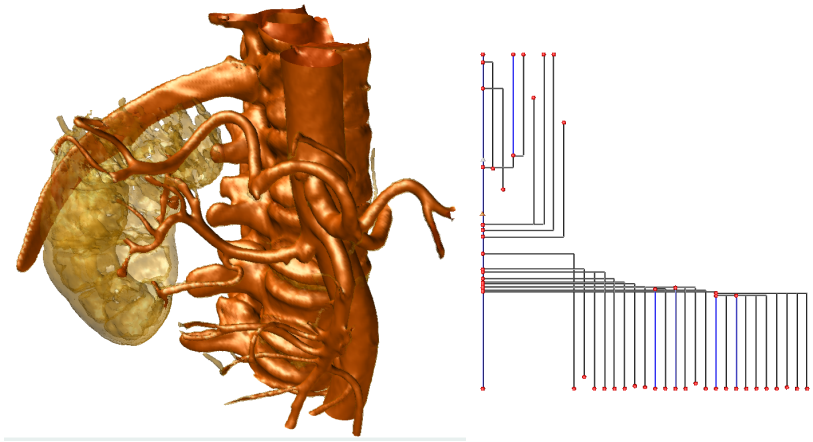


(c) engine

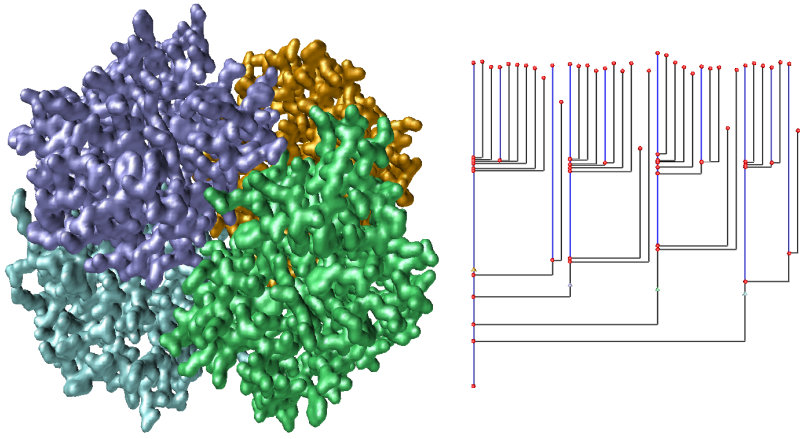
Figure 5.13: Final results



(a) visible human



(b) kidney



(c) hemoglobin

Figure 5.14: Final results

Chapter 6. Conclusions and Future Works

This thesis aim to take the advantage of modern hardware to extract the interesting information in 3D volume data. To address the problem of finding contours of interest, we construct contour tree and use it as the visual representation of the data. To apply the method to large data with noisy structures, hierarchical simplification algorithm is applied the the contour tree. Hence, an interface is created to enable the interaction to map the structure to surface mesh. The hybrid algorithm proposed in this thesis improved the mapping action significantly by threaded propagation in CPU and GPU accelerated triangulation.

The reason to use CPU for the propagating operation mainly because it is implemented with heavy conditional controls. Multi-core CPUs got its advantage to manage these operation with highly developed thread synchronization functions, which is not yet enabled in GPU programming.

In this thesis, an assumption was made that the input data should be simplicial mesh. The application is limited to result in rectangular geometry mesh.

We want to continue this work by using more statistical information in the volume data to show salient contours in an interactive way. Besides, view-dependent

method can be applied to improve the extraction speed by propagating only the surface face to the view-point. Other future improvements include the possible to implement efficient propagation method on GPU and developing proper volume dividing method to enable the application to large dataset.

Reference

[1] C.L. Bajaj, V. Pascucci, and D.R. Schikore. The contour spectrum. In *Proceedings of the 8th conference on Visualization'97*, pages 167–ff. IEEE Computer Society Press, 1997.

[2] CL Bajaj, V. Pascucci, D. Thompson, and XY Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, pages 97–104. IEEE Computer Society, 1999.

[3] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 918–926. Society for Industrial and Applied Mathematics, 2000.

[4] H. Carr, J. Snoeyink, and M. van de Panne. Simplifying flexible isosurfaces using local geometric measures. In *Proceedings of the conference on Visualization'04*, pages 497–504. IEEE Computer Society, 2004.

[5] H. Carr, J. Snoeyink, and M. van de Panne. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. *Computational Geometry*, 43(1):42–58, 2010.

[6] R.A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *ACM Siggraph Computer Graphics*, volume 22, pages 65–74. ACM, 1988.

[7] C. Dyken, G. Ziegler, C. Theobalt, and H.P. Seidel. High-speed marching cubes using histopyramids. In *Computer Graphics Forum*, volume 27, pages 2028–2039. Wiley Online Library, 2008.

[8] F. Goetz, T. Junklewitz, and G. Domik. Real-time marching cubes on the vertex shader. In *Proceedings of Eurographics*, volume 2005, page 2, 2005.

[9] M. Harris, S. Sengupta, and J.D. Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–876, 2007.

[10] NVIDIA CUDA SDK Graphics Interop. Marching cubes isosurfaces. http://developer.download.nvidia.com/.

[11] G. Kindlmann and J.W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 79–86. ACM, 1998.

[12] J. Krüger and R. Westermann. Acceleration techniques for gpu-based volume rendering. *IEEE Computer Society*, 2003.

[13] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM Siggraph Computer Graphics*, 21(4):163–169, 1987.

[14] J. Pal Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *Computer*, 27(7):45–55, 1994.

[15] V. Pascucci and K. Cole-McLaughlin. Efficient computation of the topology of level sets. In *Proceedings of the conference on Visualization'02*, pages 187–194. IEEE Computer Society, 2002.

[16] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli. Multi-resolution computation and presentation of contour trees. In *Proceedings of the IASTED conference on Visualization, Imaging, and Image Processing*. VIIP, 2004.

[17] B.S. Sohn and C. Bajaj. Time-varying contour topology. *Visualization and Computer Graphics, IEEE Transactions on*, 12(1):14–25, 2006.

[18] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[19] M. Van Kreveld, R. Van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 212–220. ACM, 1997.

[20] J. Zhou, M. Takatsuka, and University of Sydney. School of Information Technologies. *Contour tree simplification based on a combined approach*. School of Information Technologies, University of Sydney, 2008.

국문초록

본 논문은 컴퓨터보조진단분야에서 질병의 진단및 치료를 하거나 특정한 특징을 찾기 위해 의료영상의 관심영역(ROI)을 분석하는 것은 중요하다. 특 히 다양한 의료용 응용프로그램들을 이용하여 2D나 3D 이미지 데이터들로 부터 암의 경계를 찾거나 그 크기를 측정할 수 있다.

실제로 의료영상 분석시 3D 컨투어 컴포넌트의 크기가 매우 큰 경우가 많 으며 이 경우 삼각화 및 컨투어의 전파를 하는데 많은 시간을 소요된다. 본 논문은 이러한 문제들을 해결하기 위하여 컨투어 트리를 계산하고 단순화하 였으며 멀티코어CPU와 다중코어GPU를 동시에 이용한 혼합 가속화 방방을 제안하였다. 이 혼합 가속화 방법은 각각의 장점을 이용하여 관심영역을 빠 르게 추출할 수 있었다.

키워드: *멀티 쓰레딩, CUDA, 컨투어 트리, 등위면, 컨투어 컴포넌트*

ABSTRACT

In the vision of Computer Aided Diagnosis(CAD), to find and treat specific illness, ROI (Region of Interest) analyze is an important methodology for finding special characteristics. Particularly, for 2D (an image) and 3D (a volume) datasets, many medical applications are designed to draw the boundaries of a tumor for measuring its size.

In practical study of medical imaging, surface of a specific 3D contour might be very huge. And it would be time-consuming task to propagate the contour and triangulate it into regular mesh. This thesis presents a series of methods from computing the contour tree to the simplification method. And then, taking the advantage of both multi-core CPU and many-core GPU, boundary of interest can be extracted with a hybrid accelerated method.

Keywords: *Multi-threading, CUDA, Contour Tree, Isosurface, Contour Component*