FULLY AUTOMATIC UPPER AIRWAY SEGMENTATION AND SURFACING
ON A GPU FROM CONE-BEAM CT VOLUMES

A thesis presented to
the faculty of
the College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Master of Science

Michael L. Farrell
November 2009
This thesis entitled
FULLY AUTOMATIC UPPER AIRWAY SEGMENTATION AND SURFACING
ON A GPU FROM CONE-BEAM CT VOLUMES

by
MICHAEL L. FARRELL

has been approved for
the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

Maarten Uijt de Haag
Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology
Cone-beam computed tomography (CBCT) is an attractive technique for diagnostic radiology due to its short scanning time and 3-dimensional (3D) voxel-based reconstructions. The introduction of graphical computing (GPU) techniques have accelerated the reconstruction and acquisition of these 3D volumes. Clinical analysis of 2-dimensional (2D) image slices or 3D volumes can be tedious without computer-aided segmentation techniques for identification of anatomically-pertinent structures. Research has been conducted in the area of automatic airway surface extraction (from CBCT) which was implemented using traditional programming (CPU) methods. This study focuses on GPU processing and segmentation techniques specific to the area of CBCT craniofacial scans. The outcome is a fully automatic upper airway segmentation method on a GPU using NVIDIA’s Common Unified Device Architecture (CUDA).

Approved: 

Maarten Uijt de Haag

Associate Professor of Electrical Engineering and Computer Science
Acknowledgments

I acknowledge Dr. Scott Lozanoff for his unwavering support during this project. Without his support and motivation, this work would not have reached the maturity it has today. Additionally, I thank Maarten Uijt de Haag for his support during my 2 years here at Ohio University and during the thesis project.

I also acknowledge my father, an incredibly supportive figure in my life whose encouragement has led me to finish this masters program despite derailments.

I would also like to thank Dr. G.D. Singh of Biomodeling Solutions for his support in providing the data sets used for this study.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>4</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>5</td>
</tr>
<tr>
<td>List of Figures</td>
<td>8</td>
</tr>
<tr>
<td>List of Tables</td>
<td>10</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>11</td>
</tr>
<tr>
<td>1.1 GPGPU Computing Overview</td>
<td>12</td>
</tr>
<tr>
<td>1.1.1 GPU Shader Languages</td>
<td>13</td>
</tr>
<tr>
<td>1.1.2 CUDA: A general-purpose GPU solution</td>
<td>14</td>
</tr>
<tr>
<td>1.2 Cone-beam CT Overview</td>
<td>19</td>
</tr>
<tr>
<td>2 Background</td>
<td>22</td>
</tr>
<tr>
<td>2.1 Cone-beam CT and CUDA</td>
<td>22</td>
</tr>
<tr>
<td>2.2 CBCT Airway Studies</td>
<td>24</td>
</tr>
<tr>
<td>3 Methods and Materials</td>
<td>27</td>
</tr>
<tr>
<td>3.1 Sample</td>
<td>27</td>
</tr>
<tr>
<td>3.2 GPGPU Implementation: Segmentation</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1 3D Gaussian</td>
<td>29</td>
</tr>
<tr>
<td>3.2.2 Principal Component Analysis</td>
<td>34</td>
</tr>
<tr>
<td>3.2.3 Automatic Airway Detection</td>
<td>41</td>
</tr>
<tr>
<td>3.2.4 Airway Segmentation</td>
<td>44</td>
</tr>
<tr>
<td>3.2.5 Airway Surfacing</td>
<td>48</td>
</tr>
<tr>
<td>3.2.6 Cranial Surfacing</td>
<td>52</td>
</tr>
<tr>
<td>3.3 GPU Implementation: Visualization</td>
<td>55</td>
</tr>
<tr>
<td>3.3.1 2D Visualization</td>
<td>57</td>
</tr>
<tr>
<td>3.3.2 3D Visualization</td>
<td>58</td>
</tr>
<tr>
<td>4 Experimental Results</td>
<td>65</td>
</tr>
<tr>
<td>4.1 Airway Segmentation</td>
<td>65</td>
</tr>
<tr>
<td>4.2 Craniofacial Surfacing</td>
<td>65</td>
</tr>
<tr>
<td>4.3 Execution time</td>
<td>66</td>
</tr>
<tr>
<td>4.3.1 Occupancy</td>
<td>76</td>
</tr>
<tr>
<td>5 Discussion</td>
<td>78</td>
</tr>
<tr>
<td>5.1 Segmentation</td>
<td>78</td>
</tr>
</tbody>
</table>
5.2 Performance ............................................................... 80

6 Conclusions and Future Work ........................................... 84

Bibliography ..................................................................... 87
List of Figures

1.1 Typical 3D CUDA grid ........................................... 14
1.2 Example of CUDA code ........................................... 18
1.3 Cone-beam X-ray projection .................................... 20
1.4 Commercially available CBCT scanner .......................... 21

2.1 Illustration of the volume rendering technique .................. 23
2.2 2D slices of a CBCT acquisition and corresponding 3D volume (rendered by GPU) ........................................... 24

3.1 Example of a 1D gaussian filter .................................... 31
3.2 3D gaussian filter CPU implementation used as a basis for GPU implementation, derived from [1] ................................. 35
3.3 3D gaussian filter GPU kernel (pseudocode) ..................... 36
3.4 2D voxel slice with an unfiltered left half and a 3D-Gaussian filtered right half computed by GPU (filter width=4) ..................... 37
3.5 Point sample of CBCT slice with computed principal component axes 39
3.6 Covariance computation kernel (pseudocode) ..................... 40
3.7 Eigenvector computation kernel (pseudocode) [2],[3] ................ 40
3.8 Example of 1D intensity profile signal from a CBCT slice [1] .......... 42
3.9 Airway detection scan algorithm (pseudocode) ..................... 43
3.10 Output of eigenvector-oriented GPU airway detection algorithm . . . 45
3.11 Airway segmentation approach ........................................ 45
3.12 3D airway segmentation kernel (pseudocode) ......................... 46
3.13 Output of airway segmentation algorithm .............................. 47
3.15 Output of marching cubes airway surfacing ........................... 51
3.16 Cranial surfacing kernel (pseudocode) ................................ 53
3.17 Cranial surface topology .................................................. 54
3.18 Output of the craniofacial surfacing algorithm ......................... 56
3.19 2D rendering kernel (pseudocode) ..................................... 58
3.20 High and low density volume renderings ............................... 61
3.21 Volume rendering by fragment shader ................................. 61
3.22 Various 3D renderings of superimposed structures ................... 63
4.1 Airway segmentation surfaces (samples 1-9) ............................. 67
4.2 Airway segmentation surfaces (samples 9-14) ............................ 68
4.3 Craniofacial surface renderings (samples 1-6) ............................ 69
4.4 Craniofacial surface renderings (samples 6-12) .......................... 70
4.5 Craniofacial surface renderings (samples 13-14) ........................ 71
4.6 Plot of GPU time percentages for each method .......................... 75
5.1 Airway segmentation recomputation with larger gaussian filter (sam-

ples 13 and 14) .............................................................. 79
List of Tables

3.1 Sample Overview .............................................. 28

4.1 GPGPU vs CPU timing totals ................................. 72
4.2 GPGPU individual times for GeForce 9800 GT (ms) ....... 73
4.3 Individual times for CPU (ms) ................................. 73
4.4 Individual speedup factors (CPU vs GPU) ................. 74
4.5 Gaussian filter times for different filter widths (times in ms) .... 76
4.6 Multiprocessor occupancies for GPU kernels ............. 77
Chapter 1

Introduction

With cone-beam computed tomography (CBCT), clinical radiologists are presented with 3-dimensional (3D) interactive patient reconstructions in a higher-quality than previously available [5], [6]. Additionally, CBCT provides a reduced radiation exposure to patients, accurate measurements, and fast scanning times making it an attractive alternative to clinicians over traditional CT [7]. Until recently, CBCT volumes were unattainable in a short time [8]. This was due to the computationally complex nature of filtered back-projection [9], the principal algorithm used to reconstruct CBCT volumes from the raw data. With the advent of parallel graphical computing techniques, many software systems, including CBCT reconstruction systems, have benefited from substantial acceleration [10]. Furthermore, recent advances have permitted the use of standardized high-level programming languages for development within these systems. NVIDIA’s Common Unified Device Architecture (CUDA) allows programmers to use the C programming language to implement and execute routines on the graphics chip [11]. In [5], CUDA was used to implement a fast CBCT reconstruction algorithm on the graphics processor (GPU) suitable for use in real-time applications.

This work expands on GPU methods with respect to CBCT. The hypothesis to be tested is: do GPU methods with respect to CT segmentation provide significant
speed increases over the CPU with accurate results that are comparable to the non-accelerated methods. Secondly, are these results useful for diagnostic purposes in the area of obstructive sleep apnea (OSA) studies. Likewise, the goals of this study are twofold: to accurately and fully automatically detect and segment the upper airway resulting in a digital volumetric (raster) and surface (vector) reconstruction; to implement these steps on the GPU to provide sufficient performance for clinical OSA applications. The specific contribution of this work is the fully-automatic nature of the airway result which provides a robust segmentation that is free from user-bias. In chapter 3, an algorithm is presented to automatically detect, segment, and resurface the upper airway volume from 14 CBCT sample sets. The study concentrates on the upper airway which includes the oropharynx and the laryngopharynx from soft the palate to vocal cords. Principal component analysis (PCA) is applied to characterize the data as well as provide guidance for detection. Similarly, spacial-domain filtering techniques are applied at multiple stages to reduce noise and improve the integrity of the result.

1.1 GPGPU Computing Overview

Over the past two decades, interactive computer games have pushed the envelope of computer hardware capabilities. As a result, today’s graphics cards possess substantial capabilities in the areas of parallel-processing, arithmetic operations, memory size, and memory bandwidth. These advances have made graphics boards excellent
candidates for general-purpose parallel computing [12]. General-purpose GPU computing (GPGPU) allows parallel execution on the graphics processor. These computations are general-purpose in the sense that they do not have to be related to graphics applications or visualizations of any kind despite their implementation on graphics hardware. Moreover, GPGPU implementations can be dispatched remotely over the Internet and executed on the GPU without having access to a video display.

1.1.1 GPU Shader Languages

Standardized shader languages such as Cg and GLSL were the first step in flexible GPU programming. These languages allow the programmer to control specific parts of the hardware graphics pipeline such as the vertex processing stage and the fragment (raster element) processing stage. Due to the graphics hardware design, vertices and fragments were processed in parallel by the GPU, giving shader programmers their first glimpse at parallel GPU computing. Shader languages only supported a subset of the features of standard programming languages such as C. Likewise, access to graphics memory on the device was obscured by texture filtering and normalized (floating point) addressing. Furthermore, shader execution requires the existence of a graphics context. These shortcomings place GPU shader languages outside the realm of general-purpose computing.
1.1.2 CUDA: A general-purpose GPU solution

CUDA is an architecture developed by the NVIDIA corporation in response to the demand for a general-purpose parallel computing language [11]. The CUDA programming model allows programmers to use familiar high-level programming languages such as C/C++ to utilize the GPU and its on-board memory for any general-purpose computation. Before CUDA, graphical computing development required the use of limited languages such as the aforementioned high-level shader languages or arcane assembly languages. Developers may work within CUDA by using the C programming language to write kernels which are functions that execute on the graphics device, or commonly referred to as just the device. These kernels are dispatched from the host (CPU) and are able to access device memory and compute in parallel over a logical 3-dimensional thread space called a block. Multiple block’s are arranged in 1-dimensional or 2-dimensional grids. Figure 1.1 shows a logical arrangement of thread-blocks within a typical CUDA grid.

![Figure 1.1. Typical 3D CUDA grid](image-url)
During execution, each CUDA thread is permitted to access its 3D thread id, block id, and block dimension to determine its logical position within the grid. This is done using special global variables which are part of the CUDA language. Likewise the threads have access to different memory types. These memory types consist of a private memory space (per-thread), a shared memory (per-block), and a global memory space. Additionally, threads may read from two high-performance read-only memory areas: constant and texture memory. Because of the high memory bandwidth on GPU hardware, many threads can randomly access the texture memory in parallel without significant performance penalties. Unlike texture memory, the constant memory space is limited. This memory space is better suited for small constant values such as defined constants and small arrays. While constant on the device, this memory space can be programmed by the host at any time prior to a kernel launch. The shared memory space exists as a high-performance writable cache that can be used by thread-blocks to store intermediate results without having to go back to global device memory. When writing to shared memory in parallel, synchronization issues exist. To avoid such race conditions when using shared memory, CUDA provides a set synchronization primitives for use within kernels. Additionally, CUDA provides atomic functions to perform basic operations such as addition and swap during one uninterruptable step. However, these atomic functions are not available on all GPUs [11].
CUDA thread-blocks are scheduled by GPU components named Streaming Multiprocessors (SMs), each consisting of eight Scalar Processor (SP) cores. SMs employ a SIMT (single-instruction multiple-thread) architecture, which maps each instruction to multiple threads in execution, each with their own register state and program counter. To support this architecture, the GPU hardware contains a higher ratio of arithmetic logic units (ALUs) to that on the CPU. Before they are executed by the SIMT unit, thread-blocks are divided into units consisting of 32 threads called warps [11]. During runtime, the ratio of active warps to total warps (named the occupancy) is attainable as a measure of achieved parallelism for a kernel. This statistic is heavily influenced by block size.

When working with data, each CUDA thread block can be programmed to work on a subset of the data in parallel. The CUDA runtime API provides routines to aid in allocation and transfer of data to the different sections of the graphics memory. Furthermore, routines exist for logical arrangement of the data into 1D, 2D, or 3D CUDA arrays. These arrays can be associated with 1D, 2D, or 3D texture objects, respectively. Unlike traditional memory access, CUDA permits the use of normalized floating point addressing modes to access the texture objects. Similarly, when using normalized addressing, linear filtering can be enabled to provide hardware interpolation between the texture memory cells (called texels). Unlike shader languages, these memory access modes are optional. For on-screen visualization, CUDA allows kernels to directly access frame buffer memory. This is done in cooperation with popular
graphics APIs such as OpenGL and Direct3D. Lastly, CUDA allows programmers to execute linear code on the host concurrently with kernel execution on the graphics device. Specifically, series of host API and kernel calls can be rearranged into CUDA streams and executed concurrently. CUDA streams permit programmers to overlap code execution with memory transfer (IO) operations [11].

Because CUDA executes on the graphics hardware, strict limitations are imposed on the resources available to a CUDA programmer. Perhaps the most significant limitation is that of registers per block. On the most powerful hardware available for this study, the maximum register count per block is 8192. If this limit is exceeded, the kernel will fail to launch. Similarly, the running threads are limited to 16 KB of shared memory. The entire CUDA context and its running kernels are limited by the total amount of device memory (VRAM) available to the system. This includes video memory required for the display surface, desktop compositing managers, an other 3D graphics applications that may be running concurrently with the CUDA application. If this resource is exhausted, requests for allocations in device memory will fail. This is different from computation on the CPU, where hard disk swapping is performed by the operating system’s memory manager to free memory when it is scarce. Another important limitation within CUDA is that of the block size. No matter the dimensions, each CUDA block must contain no more than 512 total threads during a kernel execution [11].
Figure 1.2 shows an example of a kernel written for CUDA in C that will replace all instances of the input parameter in an array with zero. This code works by first obtaining its absolute thread index within the grid through use of the blockIdx, blockDim and threadIdx variables which are set by the CUDA API. To calculate the absolute thread index, a multiplication is performed using the builtin ulmul24 operator which performs GPU-optimized multiplication for 24-bit or smaller unsigned integer values. Once obtained, the thread index \( i \) is used for a simple comparison to replace certain values in the input array with zero as determined by the find parameter. Its useful to think of \( i \) as a loop control variable and that the kernel function is acting as the inner block of a for loop that is executed in parallel. The extra constraint \((i < sz)\) is needed to prevent over-running array bounds during conditions when the last thread block spills over the computational domain. This will always occur if the data size is not a multiple of the block size.

```c
__global__ void d_repl_value(uint *d_array, size_t sz, uint find) {
    // find threads logical position within CUDA grid
    uint i = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if (i < sz && d_array[i] == find)
        d_array[i] = 0;
}
```

Figure 1.2. Example of CUDA code

In addition to CUDA, other GPGPU technologies exist and are provided by various other commercial and non-commercial entities. Among these are IBM’s Cell architecture, ATI Stream, ClearSpeed, and OpenCL. The most attractive of these
alternatives is OpenCL which provides an open standard for GPGPU computing and can run on a broader range of hardware. The reasons for this authors selection of CUDA over OpenCL for this study stem from the lack of widespread OpenCL adoption during the early development stages of this project, and the maturity level of the CUDA API in comparison. Furthermore, the reasons for the preference of CUDA over any of the other alternatives lie in the widespread availability of NVIDIA graphics hardware.

1.2 Cone-beam CT Overview

Cone-beam computed tomography is a form of computed tomography (CT) that uses a conical-shaped X-ray beam to provide volumetric images [7]. Cone-beam CT was first presented in 1998 by [13] for maxillofacial applications. The cone-beam CT method utilizes a rotating X-ray cone-beam emitter that rotates opposite to a planar detector. This approach allows for the acquisition of volumetric data in a single X-ray scan [7], [14]. Figure 1.3 illustrates the relationship between the emitter, the cone-beam and the detector. The scanner produces raw data in the form of a series of 2D projection images which must be post-processed to obtain a volumetric reconstruction.

Cone-beam CT differs from fan-beam CT which uses a 1-dimensional detector. Because cone-beam CT provides more data than fan-beam CT in the same scanning duration, cone-beam CT reduces the scanning time and radiation exposure required to
Figure 1.3. Cone-beam X-ray projection

obtain 3-dimensional reconstructions from subjects, making it a commercially viable and safer solution for clinical use [7]. Cone-beam acquisitions are stored as digital 3D rasterized volumes. These volumes consist of discrete volume elements called voxels. CT Volumes are typically stored on a computer as series of images containing axial voxel-based slices. This storage method is useful considering the data sets may be analyzed in 2D using the individual slices [7]. Figure 1.4 shows a CBCT unit that is used in commercial practice. Cone-beam CT volumes may differ in quality as determined by the number of individual projections used to produce them. Data obtained with higher projection counts produce cleaner volumetric reconstructions that do not suffer from noise degradation and scatter artifacts from metal implants. However obtaining these higher resolution volumes requires longer scanning times and increases patients’ exposure to radiation [7].
Figure 1.4. Commercially available CBCT scanner
CHAPTER 2

Background

2.1 Cone-beam CT and CUDA

Although the scanning times for CBCT are in the durational range of 8-70 seconds[7], the raw projection data obtained from a cone-beam scanner must be transformed through reconstruction into a usable form. Since the introduction of GPU-based methods for cone-beam CT, research has been conducted in the area of GPU-accelerated cone-beam reconstructions. This work has reduced the acquisition time of a 3D patient volume from minutes to seconds [5], [8].

In 2007, [15] implemented the Feldkamp reconstruction (FDK) algorithm [9] on the GPU using a programmable shader language. Their tests ran on an NVIDIA 8800 GTX GPU. Exploring the limits of the programmable pixel and fragment processors, they achieved a 6.8 second reconstruction time of a $512^3$ volume from 360 projections of size $512^2$. Using the same graphics hardware and a similar configuration [8] implemented FDK on the GPU using CUDA in 2008. Their GPGPU approach focused on memory latency hiding techniques and a reduction of off-chip memory accesses. Using CUDA, they were able to obtain a 5.6 reconstruction time of a $512^3$ volume. These studies have extended the capabilities of cone-beam CT to include real-time clinical applications.
Unrelated to CBCT, general studies in the area of GPU techniques applied to medical imaging have been conducted. In [16], a survey is given of medical segmentation techniques and their applicability to the GPU. The authors conclude that GPU computing is suitable for interactive segmentation in which the results can be guided by user intervention in real-time. For visualization purposes, [17] presents a volume rendering method suitable for rendering segmented volumes which uses a discrete ray sampling technique implemented on the GPU. Very similar to ray-tracing, the technique follows rays originating from the image plane and accumulates the intensities at each sampling point that falls within the volume. When implemented on the GPU, this technique achieves a significant performance increase over the CPU-based method due to the highly parallel nature of ray-casting algorithms. Figure 2.1 shows an illustration of the volume rendering process.

![Illustration of the volume rendering technique](image-url)
Figure 2.2 shows the 2D axial slices and corresponding 3D volume rendering of a reconstructed CBCT volume using the methods provided in [17].

Figure 2.2. 2D slices of a CBCT acquisition and corresponding 3D volume (rendered by GPU)

2.2 CBCT Airway Studies

Recent studies have been conducted in the area of upper airway segmentation and surfacing using cone-beam CT volumes. In [18], pixel thresholding techniques were applied to automatically segment and obtain a 3D view of the upper airway. This work was implemented on the CPU, but no metrics were given for execution time. Similarly, 3D rendering was performed through use of the VTK toolkit. The given approach relies on a bounding quadrangle, which must be user-selected. While trivial, the user-selection process precludes the algorithm from being fully automatic. The implementation of pixel intensity thresholding is relatively straightforward, making it an excellent candidate for GPGPU implementations.
In [19], an algorithm was developed to segment and reconstruct the maxillary sinus from cone-beam CT. Unlike the previous approach, the algorithm uses a branching technique, requiring a user-selected starting point in each of the right and left nasal cavities. The propagation is then halted when familiar qualities of the nasal passages are identified and observed. This technique is occasionally referred to as region growing. Typically, implementation of this technique requires a recursive approach. Due to CUDA’s lack of direct support for recursion and the overall poor scaling qualities of recursive algorithms, implementing this method on the GPU is non-trivial. Secondly, for very complex and non-contiguous regions of interest, region growing becomes tedious as a starting point is required inside of each contiguous region to be filled.

An interesting algorithm presented in [20] uses gradient vector flow (GVF) snakes to detect airway contours and extract the airway surface. Airway measurements such as area and volume were also given.

In a recent work funded by the NIH, [21] devised a manual segmentation technique for imaging and surfacing the upper airway of OSA patients. This study employed the use of a user-selected boundary and a commercial software system to extract the airway volume. The volume acquisition time was given as 75-77 seconds. However, run-times for the segmentation computation and surfacing were not given. Lastly, the study provided measurements useful for OSA studies, namely the anteroposterior and lateral dimensions of the smallest cross-section area. The manual segmentation techniques presented in this work are accurate, but become infeasible for large vol-
umes of patient data. In addition to these airway-specific studies. Commercially available medical imaging software such as Amira [22] can be used to provide manual segmentation of the airway.

In previous work by this author ([1]), a fully-automatic segmentation algorithm was developed to identify the upper airway from cone-beam CT volumes with the aid of signal processing techniques. In this work, a 3D gaussian filter was used to preprocess volumes and remove x, y, and z-dependent noise. After application of the gaussian kernel, the background was segmented and eliminated. Lastly, a scanning threshold scheme along with 1D profile-analysis technique was used to identify the airway. However, without the existence of a threshold region bound (as per [18]), the algorithm was sensitive to noise despite the gaussian technique. Eigenvectors were also computed using principal component analysis (PCA), but were not used to aid in airway segmentation. These studies were implemented sequentially on the CPU using traditional programming languages.
Chapter 3

Methods and Materials

3.1 Sample

A sample of 14 slice-based CBCT image volumes were obtained from Dr. G.D. Singh, Biomodeling Solutions, Portland OR, following IRB approval. The scanned volumes consist of the craniofacial region of 14 subjects. The volumes are of varying sizes, and are stored as sets of JPEG axial image slices (one image per slice). Images were obtained using a cone-beam scanner from ICAT, Image Sciences International, Inc, Hatfield PA. For each volume, the distance between voxel-slices is 0.4 mm. Raw CT data is not available for the purposes of this study. Lastly, the sample consists of both male and female subjects.

The 14 samples are ranked in descending order from the highest to the lowest mean peak signal-to-noise ratio (PSNR) [23], [24], measured in dB. These values offer a comparable metric for image quality and noise degradation in the CBCT sets. PSNR means are obtained by computing a PSNR from 3 representative slices of the volume, and averaging the result. The individual PSNRs are computed from the root mean squared error (RMSE) between the source image \((f(\cdot))\) and a k=5 median filtered
version of the same image \( (g(\cdot)) \) as follows:

\[
MSE = \frac{1}{MN} \sum_{i}^{M} \sum_{j}^{N} [f(i, j) - g(i, j)]^2
\]
(3.1)

\[
RMSE = \sqrt{MSE}
\]
(3.2)

\[
P SNR = 20 \cdot \log_{10} \left( \frac{255}{RMSE} \right)
\]
(3.3)

Table 3.1 shows an overview of the sample sets and their sizes.

<table>
<thead>
<tr>
<th>Sample</th>
<th>Mean PSNR (dB)</th>
<th>Volume Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>37.72</td>
<td>400x400x321</td>
</tr>
<tr>
<td>2</td>
<td>37.25</td>
<td>534x534x429</td>
</tr>
<tr>
<td>3</td>
<td>36.84</td>
<td>400x400x321</td>
</tr>
<tr>
<td>4</td>
<td>36.67</td>
<td>400x400x321</td>
</tr>
<tr>
<td>5</td>
<td>36.62</td>
<td>400x400x324</td>
</tr>
<tr>
<td>6</td>
<td>35.47</td>
<td>400x400x321</td>
</tr>
<tr>
<td>7</td>
<td>35.21</td>
<td>400x400x321</td>
</tr>
<tr>
<td>8</td>
<td>34.82</td>
<td>400x400x323</td>
</tr>
<tr>
<td>9</td>
<td>34.81</td>
<td>400x400x321</td>
</tr>
<tr>
<td>10</td>
<td>34.28</td>
<td>400x400x323</td>
</tr>
<tr>
<td>11</td>
<td>33.75</td>
<td>400x400x321</td>
</tr>
<tr>
<td>12</td>
<td>32.41</td>
<td>400x400x323</td>
</tr>
<tr>
<td>13</td>
<td>29.87</td>
<td>400x400x323</td>
</tr>
<tr>
<td>14</td>
<td>29.39</td>
<td>400x400x323</td>
</tr>
</tbody>
</table>

3.2 GPGPU Implementation: Segmentation

The algorithms presented in this section are implemented using the CUDA toolkit version 2.1 in combination with the C++ language. The algorithms are executed in the order that they are presented. The methods described are executed on the GPU
unless noted otherwise noted. For each kernel, the CUDA thread-block and grid
dimensions are predetermined to provide maximum occupancy and efficiency. The
methods have been implemented and tested on an NVIDIA Quadro FX 1700 with
512 megabytes (MB) of graphics memory, and a Geforce 9800 GT with 1 gigabyte
(GB) of graphics memory.

Prior to any CUDA kernel execution, JPEG images for a single CBCT sample are
loaded from the hard disk into main memory where they are then transferred to the
device memory as logical 3D CUDA arrays. Next, the 3D CUDA arrays are bound
to 3D texture objects which can be accessed by device code inside kernels using the
modes discussed in section 1.1.2. At this point, the texture object holds the CBCT
volume in question, and the system is prepared for data processing. When referring
to the dimensionality of the current CBCT volume, $M$ refers to the width, $N$ refers
to the height, and $D$ refers to the depth. Each volume requires a significant amount
of storage in device memory ($M \times N \times D$ bytes, assuming 1 byte per voxel). Typically
this equates to roughly 50 MB per volume. During some stages of the segmentation,
multiple volumes must coexist in memory. Such a scenario occurs when writable
volumes need to be transferred to the read-only texture memory.

### 3.2.1 3D Gaussian

As shown in section 3.1, the sample volumes are of varying levels of quality due
to noise degradation. To eliminate high-frequency noise in the volumes, a low-pass
filter must be used. Because the input volume contains noise which is dependent in 3-dimensions, a 3D gaussian filter is needed to adequately eliminate the noise. To perform the gaussian smoothing, a filter is constructed with the following impulse response or kernel in 3-space as [1]:

\[ g(x, y, z) = \left( \frac{1}{\sqrt{2\pi\sigma_x}} e^{-\frac{x^2}{2\sigma_x^2}} \right) \cdot \left( \frac{1}{\sqrt{2\pi\sigma_y}} e^{-\frac{y^2}{2\sigma_y^2}} \right) \cdot \left( \frac{1}{\sqrt{2\pi\sigma_z}} e^{-\frac{z^2}{2\sigma_z^2}} \right) \]  

(3.4)

Additionally, [1] claims that such a filter is separable and thus can be represented as an iterative process of 2D and 1D convolutions, namely:

\[ h_i(x, y) = g(x, y) \ast f_i(x, y, z_i); i = 1, 2, \ldots, D \]  

(3.5)

where

\[ g(x, y) = \left( \frac{1}{\sqrt{2\pi\sigma_x}} e^{-\frac{x^2}{2\sigma_x^2}} \right) \cdot \left( \frac{1}{\sqrt{2\pi\sigma_y}} e^{-\frac{y^2}{2\sigma_y^2}} \right) \]  

(3.6)

is the 2D gaussian filter and the gaussian-filtered volume, \( Vol \), is obtained as [1]:

\[ Vol(x, y, z) = g(z) \ast \sum_{i}^{D} h_i(x, y) \]  

(3.7)

where

\[ g(z) = \frac{1}{\sqrt{2\pi\sigma_z}} e^{-\frac{z^2}{2\sigma_z^2}} \]  

(3.8)

is the 1D gaussian filter. Figure 3.1 shows an example of a 1D discrete gaussian filter.
The efficient discrete-time CPU implementation of the 3D gaussian filter is given in [1]. It is modified for clarity, and reproduced in pseudo-code in figure 3.2.

The parallel GPU implementation of the 3D gaussian filter is similar to the CPU implementation. It entails executing the code inside of the x,y,z for-loops in parallel within CUDA thread-blocks, such that each thread is responsible for filtering a single voxel. Next, the convolution passes of each filter step are separated into isolated code blocks which are conditionally executed depending on the pass parameter. This step is necessary due to the register-count limitation on some GPU hardware [11]. The pseudo-code for the GPU kernel for 3D gaussian convolution is given in figure 3.3.

To prepare space for the gaussian execution, 2 volumes of size $M \times N \times D$ are allocated in device memory: a temporary volume and an output volume. Typically, during a filter implementation, when memory is scarce and the input is replaceable, the input data is used as the temporary volume and written to as needed. This would require the allocation of only 1 volume, as the input and output volumes could take the roll of the temporary volume. However, in this scenario, the input volume is
allocated in the read-only texture memory on the GPU. This is done to accelerate the execution of the first filter pass, as texture memory accesses offer higher performance and memory bandwidth during kernel execution. Figure 3.4 shows a comparison of a CBCT volume slice before and after the application of the 3D gaussian filter.

The thread-block scheme for the gaussian filter is as follows. Threads are logically arranged into 3D blocks of size $8^3$, a value empirically discovered to maximize occupancy for this kernel. As discussed in section 1.1.2, CUDA grids may only be 1 or 2-dimensional. Thus, a volume larger than $M \times N \times 8$ cannot be filtered during a single kernel invocation. These thread-blocks are then arranged into a 2D grid capable of filtering depthwise volumetric sections of 8 slices during one kernel execution. Lastly, a z-offset value is passed to each kernel so that the it may operate on any depthwise section of the volume. To filter the entire texture volume, the gaussian filter kernel is invoked three times for each depthwise section, once for each of the $x,y,z$ filter passes. It is important to note that the kernel execution on the depthwise sections may be done in parallel through use of CUDA streams. Such a process would encompass the overlapping of I/O operations (host to device memory transfers for the input texture) with kernel execution.

It is important to select a filter width that preserves the important features in the CT data. Using a filter that is too wide can mask thin anatomical structures such as nasal passages and bone fragments. Conversely, a filter that is too thin will leave behind damaging noise, which can confuse proceeding segmentation algorithms,
resulting in false positives in the recognition. For this study, a base filter width of 2 was empirically determined, and was increased as necessary as determined by signal quality (quantitatively measured by PSNR). After the completion of the 3D gaussian process, the original texture volume is replaced with the filtered volume, and all other GPU memory is freed.

To ensure both efficiency and flexibility, gaussian filters of varying widths are precomputed and stored in host memory in one large array. Prior to the gaussian kernel launch on the GPU, the pertinent gaussian factors are copied to an array residing in constant memory on the device. This is done so that, if desired, no recompilation is required to change the default behavior of the filter at run time. Secondly, no performance penalty will be incurred by changing the filter as it will still reside in constant memory during the GPU execution.

One additional caveat arises when the gaussian filter width is small. Because of the computational power GPUs have over CPUs within their SIMT architecture, it becomes more efficient to compute the convolution directly in 3D for small filters. Though counter-intuitive, this can be seen by considering the overhead involved during each transfer of the 1D convolution results to the intermediate buffers, and waiting for each separable convolution to finish before retrieving the results and continuing to the next. By doing the entire convolution in a single kernel execution, this overhead is eliminated and performance of the 3D gaussian is increased. Thus, for small
filter widths, this direct convolution approach is used in place of the separable 1D convolution passes.

It is important to realize that while the gaussian is used to aid in the segmentation process, it does not modify the original CT data which is stored on the disk. Thus, after the segmentation is complete, the original CT volume can be loaded to restore the data to its unfiltered state.

### 3.2.2 Principal Component Analysis

Due to the human-characteristic of the sample sets, the orientation of the cranial structure is arbitrary, thus it may slightly differ from sample-to-sample. To extract the pertinent coordinate frame for segmentation, principal component analysis (PCA) is applied [25]. Specifically, major and minor axes are computed for each voxel-slice using one 2D PCA computation per slice. First, each voxel-slice is represented as an \( k \times 2 \) matrix where each column represents the position of a voxel with an intensity greater than a predetermined intensity cut-off value:

\[
X' = \begin{bmatrix}
  x_0 & x_1 & x_2 & \ldots & x_k \\
  y_0 & y_1 & y_2 & \ldots & y_k 
\end{bmatrix}
\]  

(3.9)

We compensate for the non-zero mean and define \( X \) as follows:

\[
X = \begin{bmatrix}
  (x_0 - \mu_x) & (x_1 - \mu_x) & (x_2 - \mu_x) & \ldots & (x_k - \mu_x) \\
  (y_0 - \mu_y) & (y_1 - \mu_y) & (y_2 - \mu_y) & \ldots & (y_k - \mu_y) 
\end{bmatrix}
\]  

(3.10)

Next, the covariance matrix is obtained for \( X \):
Input: $Vol_i[M, N, D]$
Output: $Vol_o[M, N, D]$
$gaus\_vector \leftarrow [1, 2, 1]$
$gaus\_sum \leftarrow 4$
$kern\_width \leftarrow 3$ //discrete filter width is 3 for this example

//first pass (filter x)..
for $z = 0$ to $D$ do
  for $y = 0$ to $N$ do
    for $x = 0$ to $M$ do
      $sum \leftarrow 0$
      for $k = 0$ to $kern\_width$ do
        $cx \leftarrow x - (kern\_width - 1)/2 + k$
        $ivalue \leftarrow Vol_i[cx, y, z]$
        $sum \leftarrow sum + ivalue \ast gaus\_vector[k]$
      end for
      $result \leftarrow sum/gaus\_sum$
      $Vol_o(x, y, z) \leftarrow result$
    end for
  end for
end for

//second pass (filter y)..
for $z = 0$ to $D$ do
  for $y = 0$ to $N$ do
    for $x = 0$ to $M$ do
      $sum \leftarrow 0$
      for $k = 0$ to $kern\_width$ do
        $cy \leftarrow y - (kern\_width - 1)/2 + k$
        $ivalue \leftarrow Vol_i[x, cy, z]$
        $sum \leftarrow sum + ivalue \ast gaus\_vector[k]$
      end for
      $result \leftarrow sum/gaus\_sum$
      $Vol_o(x, y, z) \leftarrow result$
    end for
  end for
end for

//final pass (filter z)..
for $z = 0$ to $D$ do
  for $y = 0$ to $N$ do
    for $x = 0$ to $M$ do
      $sum \leftarrow 0$
      for $k = 0$ to $kern\_width$ do
        $cz \leftarrow z - (kern\_width - 1)/2 + k$
        $ivalue \leftarrow Vol_i[x, y, cz]$
        $sum \leftarrow sum + ivalue \ast gaus\_vector[k]$
      end for
      $result \leftarrow sum/gaus\_sum$
      $Vol_o(x, y, z) \leftarrow result$
    end for
  end for
end for

Figure 3.2. 3D gaussian filter CPU implementation used as a basis for GPU implementation, derived from [1]
Input: \( Vol([M, N, D], zo) \): the slice offset, \( pass: x, y, \) or \( z \) filter pass
Output: \( Vol([M, N, D]) \)

\[
\text{gaus\_vector} \leftarrow [1, 2, 1]
\]

\[
gaus\_sum \leftarrow 4
\]

\[
kern\_width \leftarrow 3 \quad // \text{discrete filter width is } 3 \text{ for this example}
\]

\[
x \leftarrow \text{blockIndex}_0 \ast \text{blockDim}_0 + \text{threadIndex}_0 \quad // \text{obtain block and grid information from CUDA}
\]

\[
y \leftarrow \text{blockIndex}_1 \ast \text{blockDim}_1 + \text{threadIndex}_1 \quad // \text{then compute the logical thread position}
\]

\[
z \leftarrow \text{blockIndex}_2 \ast \text{blockDim}_2 + \text{threadIndex}_2 + zo
\]

\[
\text{if } pass = 0 \text{ then}
\]

\[
// x \text{ filter pass}
\]

\[
\text{sum} \leftarrow 0
\]

\[
\text{for } k = 0 \text{ to } \text{kern\_width} \text{ do}
\]

\[
\text{cx} \leftarrow x - (\text{kern\_width} - 1)/2 + k
\]

\[
\text{ivalue} \leftarrow Vol([\text{cx}, y, z]) \quad // \text{texture access}
\]

\[
\text{sum} \leftarrow \text{sum} + \text{ivalue} \ast \text{gaus\_vector}[k]
\]

\[
\text{end for}
\]

\[
\text{result} \leftarrow \text{sum}/\text{gaus\_sum}
\]

\[
Vol_o(x, y, z) \leftarrow \text{result}
\]

\[
\]

\[
\text{else if } pass = 1 \text{ then}
\]

\[
// y \text{ filter pass}
\]

\[
\text{sum} \leftarrow 0
\]

\[
\text{for } k = 0 \text{ to } \text{kern\_width} \text{ do}
\]

\[
\text{cy} \leftarrow y - (\text{kern\_width} - 1)/2 + k
\]

\[
\text{ivalue} \leftarrow Vol([x, cy, z])
\]

\[
\text{sum} \leftarrow \text{sum} + \text{ivalue} \ast \text{gaus\_vector}[k]
\]

\[
\text{end for}
\]

\[
\text{result} \leftarrow \text{sum}/\text{gaus\_sum}
\]

\[
Vol_t(x, y, z) \leftarrow \text{result}
\]

\[
\]

\[
\text{else if } pass = 2 \text{ then}
\]

\[
// z \text{ filter pass}
\]

\[
\text{sum} \leftarrow 0
\]

\[
\text{for } k = 0 \text{ to } \text{kern\_width} \text{ do}
\]

\[
\text{cz} \leftarrow z - (\text{kern\_width} - 1)/2 + k
\]

\[
\text{ivalue} \leftarrow Vol([x, y, cz])
\]

\[
\text{sum} \leftarrow \text{sum} + \text{ivalue} \ast \text{gaus\_vector}[k]
\]

\[
\text{end for}
\]

\[
\text{result} \leftarrow \text{sum}/\text{gaus\_sum}
\]

\[
Vol_o(x, y, z) \leftarrow \text{result}
\]

\[
\]

end if

Figure 3.3. 3D gaussian filter GPU kernel (pseudocode)
Figure 3.4. 2D voxel slice with an unfiltered left half and a 3D-Gaussian filtered right half computed by GPU (filter width=4)
\[ C_X = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{yx} & \sigma_y^2 \end{bmatrix} \]

\[ = \frac{1}{k} X X^T \] (3.11)

The 2D PCA problem is formally defined as the solution to [26]:

\[ P^T C_X P = \Lambda = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \] (3.12)

Above, the columns of P represent the eigenvectors or principal components, whereas \( \lambda_{1,2} \) are the respective eigenvalues. The solution to this problem can be provided through eigenvector decomposition. For the 2 \times 2 case, a solution is provided in [3].

It is useful to visualize \( X \) as a 2D grid of points corresponding to the CBCT slice; a discrete function that exists only where physical matter existed in the volume. Figure 3.5 shows a depiction of \( X \) and the computed principal component axes (major and minor) for a given CBCT voxel slice.

PCA ensures a coordinate frame that captures the largest variance through the data [25]. This is a useful result due to the large fluctuations in voxel intensities during a boundary between cranial tissue and the airway. This increases the probability that the airway lies along one of the major or minor axes obtained during the PCA. In previous work, a 3D PCA was suggested alongside the 2D PCA [1]. Unlike 3D PCA, the slice-based 2D PCA captures the dynamics of the volume and airway gap through
Figure 3.5. Point sample of CBCT slice with computed principal component axes

the progression down the $z$ axis. Applying a 3D PCA to the craniofacial volume may not provide enough information to detect the airway cavity. However, a 3D PCA may be applied to the segmented airway volume for statistical and analytical purposes.

The GPU implementation of PCA involves a three-step process. This process is computed in parallel over a linear $D$-sized thread-block, with each thread operating on its own 2D CBCT slice. First, means are computed for the slice in a straightforward manner. Next, the covariance matrix is computed using the pseudocode shown in figure 3.6. To complete the PCA, eigenvectors are computed using a $2 \times 2$ eigenvector decomposition algorithm from [2] based on [3]. This algorithm is given in figure 3.7. Once obtained, the eigenvectors are sorted by their eigenvalues and stored in arrays residing in global device memory for later retrieval.
Input: $Vol[M, N, D]$, $\mu[D]$: computed means, $co$: the intensity cutoff value
Output: $cov[2][2]$: the covariance

$k \leftarrow 0$
$cov \leftarrow [0, 0, 0, 0]$
$z \leftarrow blockIndex_0 * blockDim_0 + threadIdx_0$ //obtain logical thread position from CUDA

for $y = 0$ to $N$
  for $x = 0$ to $M$
    if $Vol[x, y, z] > co$
      $cx \leftarrow x - \mu[z]_x$
      $cy \leftarrow y - \mu[z]_y$
      $cov[0][0] \leftarrow cx * cx$
      $cov[0][1] \leftarrow cx * cy$
      $cov[1][1] \leftarrow cy * cy$
      $k \leftarrow k + 1$
    end if
  end for
end for
$cov[0][0] \leftarrow cov[0][0] / (k - 1)$ //normalize
$cov[0][1] \leftarrow cov[0][1] / (k - 1)$
$cov[1][1] \leftarrow cov[1][1] / (k - 1)$
$cov[1][0] \leftarrow cov[0][1]$ //symmetry in off-diagonals

Figure 3.6. Covariance computation kernel (pseudocode)

Input: $cov[2][2]$

$A \leftarrow cov[0][0]$  $B \leftarrow cov[0][1]$  $C \leftarrow cov[1][0]$  $D \leftarrow cov[1][1]$

if $B * C \leq 0.1 * 10^{-20}$ then
  $\lambda[0] \leftarrow A \lambda[1] \leftarrow D$
  $V[0] \leftarrow [1, 0]$  $V[1] \leftarrow [0, 1]$
end if
$tr \leftarrow A + D$
$det \leftarrow A * D - B * C$
$S \leftarrow \sqrt{tr * tr / 4 - det}$
$\lambda[0] \leftarrow tr / 2 + S$
$\lambda[1] \leftarrow tr / 2 - S$
$tmp \leftarrow \max[(A - D) * (A - D)) / 4 + B * C, 0]$
$SS \leftarrow \sqrt{tmp}$
if $(A - D) < 0$ then
  $V[0]_x = C$
  $V[0]_y = -(A - D) / 2 + SS$
  $V[1]_x = (A - D) / 2 - SS$
  $V[1]_y = B$
else
  $V[1]_x = C$
  $V[1]_y = -(A - D) / 2 - SS$
  $V[0]_x = (A - D) / 2 + SS$
  $V[0]_y = B$
end if
$V[0] \leftarrow V[0] / \|V[0]\|$ //normalization
$V[1] \leftarrow V[1] / \|V[1]\|$

Figure 3.7. Eigenvector computation kernel (pseudocode) [2],[3]
On the surface, the eigenvectors provide the most useful information, however, it is important not to overlook the eigenvalues. A major limitation of PCA surfaces when the computed eigenvalues are similar [25]. This signifies a symmetry within the data, indicating that there are no true principal components, or infinitely many depending on the perspective taken. This scenario occurs when the slices analyzed have a largely circular shape and the airway is small.

### 3.2.3 Automatic Airway Detection

As a first step in the airway segmentation process, the airway location needs to be detected. Moreover, the airway center should be detected as accurately as possible so airway dimensions can be estimated for the segmentation algorithm. The detection is aided by the eigenvector-oriented coordinate frame obtained from PCA in section 3.2.2; upon detection, the airway position can be expressed as a linear combination of the orthogonal eigenvectors. Assuming the airway lies along the major axis, the airway detection for a given voxel-slice \( z \) can be reduced to a profile scan of a discrete-time 1D image signal [27]:

\[
x_z(n) = Vol([0.5 + \mu(z)_x + nV_x], [0.5 + \mu(z)_y + nV_y], z); \quad n = 1, 2, \ldots, D \quad (3.13)
\]

where \( Vol \) is the CBCT volume, \( z \) is the current voxel-slice, \( \mu \) is the origin and \( V \) is the eigenvector. The 1D \( x(n) \) is depicted in figure 3.8.

Using a profile scan method similar to [1], \( x(n) \) is searched for a large and sustained intensity drop signifying entrance to the airway region-of-interest (ROI). At the airway
edge, the position $n$ is noted as the entry point. Likewise, upon exiting the airway, the exit position is recorded. Once the scan is finished, the midpoint between the entry and exit points is taken as the airway position, which can be expressed in 2D in terms of the eigenvector $\mathbf{V}$. The entry and exit points for the scan supportively provide useful information in that they can be used to estimate the size of the airway. This airway size estimate is a crucial element of the airway segmentation.

To implement the airway detection algorithm on the GPU, each scan is completed in parallel inside of a CUDA thread. To begin, a 1D thread-block of size $D$ is constructed. If $D > 512$, as per CUDA block size limitations, a 1D CUDA grid of dimensionality $\lceil \frac{D}{512} \rceil$ is constructed to handle the excess slices. Otherwise, the entire volume can be processed in one kernel execution. As shown in table 3.1, all of the samples except sample 2 fit this description. The simplified pseudo-code for this algorithm is given in figure 3.9.
Input: \( V[M, N, D], V[2] \): the eigenvectors in major/minor order, \( t_a \): airway intensity threshold, \( s_a \): airway size threshold, \( s_n \): noise size threshold, \( \mu \): trace origin

Output: \( A \): 2D airway origins

\[
\begin{align*}
z & \leftarrow \text{blockIndex0} \ast \text{blockDim0} + \text{threadIndex0} \quad \text{//obtain logical thread position from CUDA} \\
p_{\text{pos}} & \leftarrow 0 \\
t_l & \leftarrow 0 \\
a_{\text{begin}} & \leftarrow 0 \\
a_{\text{end}} & \leftarrow 0
\end{align*}
\]

for \( n = -M/2 \) to \( M/2 \) do

\[
\begin{align*}
\text{voxel} & \leftarrow x_z(n) \quad \text{//texture access} \\
\text{//out} & \Rightarrow \text{in} \\
& \text{if pos} = 0 \text{ and voxel} > t_a \text{ then} \\
& \text{pos} \leftarrow 1 \\
& \text{else if pos} = 1 \text{ then} \\
& \text{if voxel} \leq t_a \text{ then} \\
& \quad t_l \leftarrow t_l + 1 \\
& \quad \text{if } t_l \geq s_a \text{ then} \\
& \quad \text{//in} \Rightarrow \text{airway} \\
& \quad \text{pos} \leftarrow 2 \\
& \quad a_{\text{begin}} \leftarrow n \\
& \quad t_l \leftarrow 0 \\
& \quad \text{end if} \\
& \quad \text{else} \\
& \quad t_l \leftarrow 0 \\
& \quad \text{end if} \\
& \text{else if pos} = 2 \text{ then} \\
& \text{if voxel} > t_a \text{ then} \\
& \quad t_l \leftarrow t_l + 1 \\
& \quad \text{if } t_l \geq s_n \text{ then} \\
& \quad \text{//airway} \Rightarrow \text{back in} \\
& \quad a_{\text{end}} \leftarrow n \\
& \quad \text{break} \quad \text{//break out of for loop} \\
& \quad \text{end if} \\
& \quad \text{end if} \\
& \text{end if} \\
& \text{end for}
\end{align*}
\]

\[
\begin{align*}
d & \leftarrow a_{\text{begin}} + (a_{\text{end}} - a_{\text{begin}})/2 \\
A_z & \leftarrow \mu_z + dV_z[0]
\end{align*}
\]

Figure 3.9. Airway detection scan algorithm (pseudocode)

After the first scan, a second scan is performed to adjust the airway position using the minor axis. The second scan uses the same approach as the first with the exception that the second scan uses the output of the first as the origin. On the actual GPU implementation, both scans are rolled into one kernel. During the unlikely scenario presented in section 3.2.2 when the eigenvalues are similar, the profile scan can miss
the airway and the detection can fail. In this case, extra work needs to be taken to find the airway. Because of the symmetry implied by the breakdown of PCA, one solution is to perform the scan over all radials from the mean until the airway is found. This approach is sensitive to noise and therefore does require a sufficient filter width during the 3D gaussian step. Another solution is to take the mean as the airway position, relying on the assumption that the airway lies close to the mean during these cases. The latter solution is empirically found to be true although both solutions should be explored to maximize robustness of the detection method.

As a final step, the airway positions are smoothed by a 1D gaussian filter in the $z$ direction to reduce the noise. This imposes a relationship between the airway positions and their neighboring slices and adds a degree of robustness to the segmentation. While a gaussian filter cannot completely eliminate outliers within the slices, it minimizes their negative effect on the computation. If needed, other statistical methods can be utilized to reject the outliers altogether. Once computed, the per-slice airway positions remain in device memory as an input to the airway segmentation step. Figure 3.10 shows two results from the GPU airway detection algorithm.

### 3.2.4 Airway Segmentation

With the airway position computed, the airway segmentation can begin. From a high level, the airway segmentation consists of a bounded voxel thresholding technique. Like the airway detection, the segmentation is fully automatic and requires
no user intervention or correction to function. However if desired, the observer can modify segmentation thresholds to target different surfaces of interest. By default, the airway threshold parameter is set to an intensity of 5, a value determined empirically to provide the best results. For the segmentation bound, an eigenvector-oriented elliptical region is used. The dimensions of the region are obtained by the size estimates computed during the airway detection step in section 3.2.3. Figure 3.11 shows an overview of the airway segmentation process.
In order to implement the GPU segmentation, an empty volume of size $M \times N \times D$ is allocated in device memory to hold the segmentation result. To compute in a 3D logical space, a CUDA thread block is constructed of size $8^3$. Similarly, a grid is constructed with a size capable of thresholding the airway volume. Each thread computes its position from the eigenvectors, does a boundary check, and thresholds a single voxel in parallel. Figure 3.12 shows a pseudocode implementation of the segmentation algorithm and figure 3.13 shows four result slices from a volume.

Figure 3.12. 3D airway segmentation kernel (pseudocode)

While the segmentation voxels can hold 256 unique values, only two of them are significant. These values are 255 which is the marker for the segmented airway volume and 0 which represents the airway complement (everything that is not the airway such as the background and cranial matter). After the segmentation is complete, the airway volume is converted into a 3D texture in device memory. This is done to maximize efficiency during kernel accesses to the volume which are needed for
surfacing and rendering. Secondly, this allows the algorithms to take advantage of hardware filtering modes and normalized addressing as discussed in section 1.1.2. It is important to note that at this point, the airway is segmented only as a 3D raster and no surface exists.
3.2.5 Airway Surfacing

The airway surfacing step is the final crucial step in the segmentation process. Because the surface will be utilized for various forms of analysis, it must be computed in a consistent fashion without polygonal cracks or errors of any kind. For these reasons, the marching cubes algorithm is selected for this process. First proposed in 1987 by [28], marching cubes computes isosurfaces from 3D scalar fields. Marching cubes provides a stable and robust solution for extracting the airway surface from the segmented airway volume. Secondly, the algorithm is inherently parallel, making it an excellent candidate for implementation on the GPU with CUDA.

The marching cubes method involves traversing a 3D scalar field function while sampling at 8 neighboring locations. These 8 locations form a cube which is triangulated based on the sampled values to approximate the desired isosurface. The author [28] discovered that only $2^8 = 256$ possible triangulations exist within the cube. This is true because each cube vertex can either intersect or not intersect the isosurface, resulting in 256 possible combinations for the 8 samples. Each of the 256 unique cases correspond to a precomputed cube configuration of triangles. These precomputed configurations are stored in a lookup table. During the cube traversal, the field is sampled at each of the 8 cube locations to determine if it intersects the isosurface. This is done by comparing the sampled intensity to a predetermined iso value parameter. For each location that falls within the isosurface, a corresponding bit is set within an 8-bit number. For the locations that fall outside the surface, the bit is set
to 0. Once computed, the number itself becomes the index into the geometry lookup table.

The author [28] discovered a symmetry in the possible cube configurations. It was discovered that all of the 256 configurations can be reduced to 15 unique cases whose symmetrical rotations and reflections can produce the original set. Figure 3.14 from [4] depicts these 15 cases.

Figure 3.14. 15 base triangulation configurations for marching cubes [4]

When adding the vertices and triangles to the output, an interpolation step is necessary to ensure accurate positions with respect to the surface. Specifically, vertex positions are linearly interpolated along the corresponding cube edge using the sampled field intensities as an interpolation factor. While this interpolation step may introduce a bias in the quantifiable properties of the airway surface, it is important to note that the surfacing step is used strictly for visualization purposes and the rastrized volume serves as a better candidate for quantification. Once generated, the isosurface is assumed to be of constant density. In [28], it is shown that for constant-density surfaces, the gradient vector can be used to find the normal vector of the the surface.
at a given point. Analytically, the gradient vector can be obtained by differentiation of the volume density function $f$ [28]:

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

These normal vectors are used to perform lighting calculations during 3D rendering.

In the raster or voxel case, obtaining $\mathbf{g}$ analytically may not be possible. In this case, a more direct approach can be applied to each triangle as it is generated by the marching cubes algorithm. At each face, a cross product is computed between two edges of the triangle. These cross products produce vectors that are normal to the triangle surface. However, because this method yields one normal for three points on the surface, the normals will appear flat. To address this, smoothing is performed for each point by averaging all of the normals of the triangles referencing the point [29].

Two interesting developments arise from implementing the marching cubes algorithm on the GPU. First in [28], the original method stated that no more than 4 slices are required to exist in memory at one time during the traversal. However, GPUs with large amounts of memory permit the entire volume and surface geometry to be resident in memory during the computation. Secondly, the parallel nature of GPGPU computing allows each cube tessellation to be computed in parallel inside of a CUDA thread. This author is not the first to notice the potential for marching cubes on the GPU [30], [31]. With the release of the NVIDIA CUDA SDK, a marching cubes implementation for CUDA is provided. For the purposes of this study the
GPU implementation, while originally written for small volumes of size $64^3$, provided the necessary results with only minor modifications. Figure 3.15 shows the output of the airway surfacing step.

Figure 3.15. Output of marching cubes airway surfacing
For surfacing the airway, it can be observed that the isosurface exists where and only where the airway exists. Thus, the isovalue used is that of the airway marker (255) set during the segmentation process in section 3.2.4. If surfacing the entire cranial volume using marching cubes is desired, a particularly low isovalue such as the airway threshold from the preceding section will produce the desired results.

3.2.6 Cranial Surfacing

In order to interpret the surface results of the airway segmentation process, a GPU algorithm was developed to compute and extract the craniofacial surface from the CBCT volume. Unlike the airway segmentation method, this approach directly computes a 3D polygonal surface from the input volume, skipping the middle step of rasterization. This is done because of the simple nature of the background (pure black), the single-body characteristic of the ROI, and the complexities and expensive nature of marching cubes which is not always necessary. This section presents a simple approach based on ray-casting that provides a cranial surface with smooth normals in a computation time comparable to that of marching cubes when implemented on the GPU.

The cranial surfacing algorithm uses a ray-casting approach that works in a cylindrical coordinate frame. For each slice, radials are casted from the volume center in a sun ray pattern and traced inwards from empty space until they reach the cranial volume where the computation halts. The intersection distance $r$ is noted for each
radial angle $\theta$ on each slice $z$. Once converted back to cartesian coordinates, these points are stored into a vertex array. Thus, each incident ray represents a single point of the cranial surface. Because $\theta$ and $r$ are discrete, resolution parameters $T$ and $R$ are used to determine how many radials and how much sampling along each radial is performed.

Within CUDA, each radial inside of each slice can be traced in parallel due to their independent nature. Using a logical computation grid of size $T \times D$, the kernel can process the entire volume in one invocation. In this kernel, the first dimension of the thread index is used as the polar coordinate angle while the second refers the slice. Each radial is then traced using an iterative approach similar to that of standard ray-casting [32] but in the plane. The GPU algorithm is outlined in figure 3.16.

\begin{verbatim}
Input: Vol[M, N, D], Tc: cranial intensity threshold
Output: P[T, D]: traced vertices (cartesian)
$\theta_i \leftarrow$ blockIdx.x * blockDim + threadIdx
$z \leftarrow$ blockIdx.y * blockDim + threadIdx
for $r_i = 0$ to $R$
  $r \leftarrow M \times (1.0 - r_i/R)$
  $\theta \leftarrow 2\pi \times (\theta_i/T)$
  $x \leftarrow M/2 + \sin(\theta) \times r$
  $y \leftarrow N/2 + \cos(\theta) \times r$
  voxel $\leftarrow$ Vol[x, y, z] //texture access
  if voxel > Tc then
    $P[\theta_i, z] \leftarrow (x, y)$
  end if
end for
\end{verbatim}

Figure 3.16. Cranial surfacing kernel (pseudocode)

As with all arrays in C, the array $P$ is stored linearly in memory such that all of the rows within each slice are adjacent with respect to the row size ($T$ in this case). This means that to use a 2D index into $P$, it must be converted to a 1D
index as $i = z \ast T + \theta_i$. If examined closely, an underlying topology can be seen within the vertex array outputted by the radial trace. It can be seen that in all cases except for points near the pole ($\theta = 0$), for radial $P[\theta_i]$, $P[\theta_i + 1]$ is the next nearest point on the current slice. Similarly, it can be seen that $P[\theta_i + T]$ is the neighboring point directly below $P[\theta_i]$ on the next slice and $P[\theta_i + 1 + T]$ is the point directly below $P[\theta_i + 1]$. These four vertices form a quadrilateral panel at $P[\theta_i]$ that can be triangulated. Figure 3.17 illustrates this relationship.

![Figure 3.17. Cranial surface topology](image)

Using this topology, each panel can be triangulated in parallel within a CUDA surfacing kernel. Smooth normals are then computed using the methods discussed in section 3.2.5. The geometry is accumulated and stored inside of a vertex buffer object (VBO) which resides on the graphics card. The VBO can be called upon during the visualization phase by OpenGL to render the 3D surface. Additionally, it is worth noting that $\theta$ and $z$ form a parameterization of the cranial surface. These
parameterizations are useful for applying texture maps, and computing consistent localized coordinate frames (tangents and binormals) for shading applications. Figure 3.18 shows the graphical results of the craniofacial surfacing step at various resolution levels. It can be seen from these images that high resolution surfacing captures more characteristics while low resolution surfacing reduces noise.

Originally, only the point cloud associated with the vertex array was the output of the GPU algorithm, and the triangulation was left to the CPU to take advantage of advanced C++ data structures for building geometries. However, to take full advantage of GPU acceleration, the triangulation step was moved into the kernel such that the output of the GPGPU computation is a tessellated cranial surface with smooth surface normals. A different 3D approach was considered initially for tracing. This approach made use of spherical coordinates where incident rays were traced from all directions on the sphere. However, the distortion introduced by projecting or mapping [33] the cranial surface to the sphere was undesirable. The current approach assures a uniform sampling of the data along the radials.

3.3 GPU Implementation: Visualization

The kernels in this section were developed for visual verification purposes. To prevent an interdependency of the preceding GPGPU algorithms with the existence of a graphics display, the GPU techniques used for visualization are decoupled from the general-purpose solutions. No code in the prior GPGPU segmentation algorithms
(a). High resolution craniofacial surface \((T = 256)\)

(b). Low resolution craniofacial surface \((T = 128)\)

(c). Low resolution wireframe of cranial surface

Figure 3.18. Output of the craniofacial surfacing algorithm

rely on the rendering kernels presented here. This is done so that GPU hardware used to produce segmentation results can be separated from hardware used to display them. Similarly, the graphics hardware used to perform the visualizations can be less powerful than the hardware that generated them, though CUDA capability is still required. Despite their decoupling, these visualizations are crucial to the segmentation process. Without an accurate, efficient, and meaningful depiction of the volume data
and segmented airway, the human observer could become confused or misled. In the clinical case, such a result could be catastrophic.

This section assumes the 3D gaussian filtered CBCT volume and segmented airway volume are loaded into 3D textures on the device. Similarly, it is assumed that all geometry from preceding sections including the cranial surface and airway surface geometry are accessible. To prepare the system for rendering, the SDL library is used to set up an OpenGL graphics context for visual results. Next, OpenGL framebuffer objects (FBOs) are created for rasterizations and VBOs are created to hold 3D surface geometry as needed. Because CUDA is permitted to directly access the frame buffer, all rasterization steps can be performed by CUDA in a single kernel execution. This is an example of GPGPU technology being used for graphical purposes.

### 3.3.1 2D Visualization

In the 2D case, only one slice can be viewed at a time. The observer uses the input device to navigate between voxel-slices. The rendering of a CBCT voxel slice is trivial as it merely copies the texture slice into the framebuffer. Thread-blocks of size $16^2$ are created and arranged into a 2D grid which covers the $M \times N$ slice size. Each thread reads the texture value and stores it into the framebuffer. During the same step, the thread checks the airway texture and, if necessary, adds a bright red color to the framebuffer. The 2D rendering kernel is shown in figure 3.19. After the rasterized framebuffer is drawn, the 2D vector results comprising the orthogonal PCA
coordinate system and cranial surface contour are overlayed using OpenGL. Typically, CUDA is not required to perform simple 2D rasterizations such as displaying a texture as OpenGL texturing provides the necessary capability. However, the reasons for not doing so are twofold. First, the use of CUDA to compute the gaussian filter and segmentation requires storage of the volume in CUDA texture. Currently, conversion of these CUDA texture objects to OpenGL textures is not straightforward and could require more storage space on the device. Second, some OpenGL contexts require texture dimensions to be powers of two which complicates the process involved in rendering images that do not fit within this constraint. CUDA textures have no such limitation.

**Input:** *Vol*[M, N, D]: Original volume, *Vol*[M, N, D]: Segmented airway volume, w: current slice

**Output:** *F*: framebuffer output

\[
x ← blockIndex₀ \times blockDim₀ + threadIdx₀
\]

//obtain block and grid information from CUDA

\[
y ← blockIndex₁ \times blockDim₁ + threadIdx₁
\]

//then compute the logical thread position

\[
u ← x - FW/2 - M/2
\]

\[
v ← y - FH/2 - N/2
\]

//texture access

\[
voxel ← Vol[u, v, w]
\]

//texture access

\[
av ← Vol[a, u, v, w]
\]

if \(av ≠ 0\) then

\[
F[x][y] ← red
\]

else

\[
F[x][y] ← voxel
\]

end if

---

**Figure 3.19. 2D rendering kernel (pseudocode)**

### 3.3.2 3D Visualization

In 3D mode, the entire CBCT volume is visible at once. The 3D rendering can contain a combination of vector and raster graphics, all rendered by the GPU. At
this stage, it is worth noting that no additional geometry is sent over the graphics
bus to the device; all of the surfaces and rasters needed to visualize the results are
already resident on the graphics card in the form of VBOs and texture units. At high
resolution, the geometry computed from a 400x400x320 volume comprises hundreds
of thousands of polygons. Keeping these geometries stored on the device improves
rendering performance.

3D Volumetric Rendering

The 3D volume rendering method presented here is similar to the approach pre-
presented in [17]. The CBCT volume renderer uses a modified volume rendering example
from the NVIDIA CUDA SDK code sample base. The algorithm permits a variable
density parameter, a very useful tool when examining the air space and segmentation
results. Generally, ray-casting algorithms cast rays from the camera, requiring the
expression of the casted ray in world coordinates. For this reason, an inverse view
transformation is required by the algorithm as an input parameter. This transforma-
tion takes the form of a $4 \times 4$ matrix in the perspective projection case and a $3 \times 4$ in
the orthographic case. Using the same block and grid dimensions of the 2D renderer,
each fragment color is computed in parallel inside of a CUDA thread.

Within the kernel, each ray is oriented through the eye-space to world-space trans-
formation and casted into 3D space. Upon intersection with the viewable volume, the
ray is then marched from the back to the front (towards the viewer) while accumulat-
ing intensity from the volume. Density is simulated by affecting the opacity of each voxel during the accumulation. As a modification to the original code, the segmented airway volume is optionally sampled during the traversal, providing a bright red visualization of the airway which shows through the CT volume. Once the traversal is finished, the final computed color is stored in the framebuffer object (FBO) which is then displayed using OpenGL. The CUDA code for the volume renderer is available as part of the NVIDIA CUDA SDK [11]. Figure 3.20 shows two example volume renderings of a CBCT acquisition rendered with CUDA.

As previously stated, while the computational part of the process relies on a high-end graphics card, the visualization should be available to low-end machines. Thus, a more general volume rendering solution is needed that does not require a CUDA-capable device. To maintain the utilization of the GPU without using CUDA, the volume rendering algorithm was converted to GLSL in the form of a fragment shader. A volume rendering performed by the GLSL fragment shader is shown in figure 3.21. The images produced by the shader are comparable to their CUDA-rendered counterparts. However some inaccuracies are visible in the GLSL renderings. In the unlikely case that no CUDA or shader capability is available at all, a non-interactive volume rendering can be performed through the use of CUDA emulation techniques [11].
Figure 3.20. High and low density volume renderings

Vector-Raster Superimposition

A particularly useful 3D visualization involves the superimposition of the various rasterized and vector-based surfaces computed by the system. This effect is achieved by rendering the outermost layer (craniofacial surface) at a reduced opacity so that
the inner structures such as the CT volume and airway surface can be seen [34]. These effects require no additional GPGPU computation under CUDA and can be achieved using the OpenGL rendering pipeline and the programmable fragment processor. Figure 3.22 shows various 3D renderings using superimposition methods.

**Quaternions**

For visual synchronization purposes, the volumetric rasterizations and vector geometry are processed by the same linear transformations. These transformations include viewing transformations in the form of translations and rotations. To provide a trackball rotation interface, the rotations in 3D are performed using quaternions. Quaternions span a 4-dimensional (4D) space \( \mathbb{H} \), and adhere to the following multiplication rule [35]:

\[
i^2 = j^2 = k^2 = ijk = -1
\]  

(3.15)

where \( i, j, \) and \( k \) are imaginary and orthogonal. From equation 3.15, the following products are obtained [35]:

\[
ij = k \quad jk = i \quad ki = j
\]

\[
ji = -k \quad kj = -i \quad ik = -j
\]

(3.16)

Quaternions are often expressed as a 4-tuple [36]:

\[
q = q_0 + iq_1 + jq_2 + kq_3
\]

(3.17)
Figure 3.22. Various 3D renderings of superimposed structures

(a). Craniofacial surface with raster volume

(b). Craniofacial surface with airway surface

(c). Craniofacial wireframe with airway surface

(d). Craniofacial surface with raster airway segmentation

(e). Craniofacial wireframe with raster airway segmentation
where \( q_0 \) is the scalar part and \( q_1, q_2, \) and \( q_3 \) make up the vector part. From equation 3.16, the following quaternion product operation is derived as [36]:

\[
pq = p_0q_0 - \mathbf{p} \cdot \mathbf{q} + p_0\mathbf{q} + q_0\mathbf{p} + \mathbf{p} \times \mathbf{q}
\]  

(3.18)

These products between quaternions correspond to rotations in \( \mathbb{H} \). For application in 3D graphics, it’s useful to operate on unit-length quaternions (\( \|\mathbf{q}\| = 1 \)) which span the hypersphere \( S^3 \subset \mathbb{H} \) [35]. In [37], it is shown that a unit quaternion can be constructed to perform a rotation in \( \mathbb{R}^3 \) about any axis \( \mathbf{u} \):

\[
\mathbf{q} = \cos \frac{1}{2} \theta + \sin \frac{1}{2} \theta \mathbf{u}
\]

(3.19)

\[
= e^{\frac{1}{2} \theta \mathbf{u}}
\]

Such a rotation can be represented with a \( 3 \times 3 \) transformation matrix converted from quaternion \( \mathbf{q} \) of the form [37]:

\[
T = \begin{bmatrix}
1 - 2q_2^2 - 2q_3^2 & 2q_1q_2 + 2q_0q_3 & 2q_1q_3 - 2q_0q_2 \\
2q_1q_2 - 2q_0q_3 & 1 - 2q_1^2 - 2q_3^2 & 2q_2q_3 + 2q_0q_1 \\
2q_1q_3 + 2q_0q_2 & 2q_2q_3 - 2q_0q_1 & 1 - 2q_1^2 - 2q_2^2
\end{bmatrix}
\]

(3.20)

Unit quaternions have the property of a simplified inverse. Given a unit quaternion \( \mathbf{q} \), the inverse is given as [35]:

\[
\mathbf{q}^{-1} = \bar{\mathbf{q}}
\]

(3.21)

Where \( \bar{\mathbf{q}} \) denotes quaternion conjugation. This property is important, as the algorithm presented in section 3.3.2 requires an inverse view transformation as its input.
Chapter 4

Experimental Results

4.1 Airway Segmentation

Using the methods described in chapter 3, automatic airway segmentation was performed for each of the 14 samples. The segmentation results are shown in figures 4.1 and 4.2. Renderings were performed using blender [38], an open-sourced ray tracer. The segmentation software produces a 3D surface file for the airway which is saved to the hard disk in the OBJ file format. The OBJ file can be imported into 3rd party software for further analysis. For the high resolution settings, these generated meshes contain hundreds of thousands of polygons. It is useful to recall that the samples are ranked in descending order by PSNR using the statistics computed in section 3.1.

4.2 Craniofacial Surfacing

Like the airway surface, the craniofacial surfaces are generated and stored to the hard disk using the OBJ format. The high quality renderings of these surfaces is given in figures 4.3 through 4.5. These meshes were post-processed using a surface smoothing tool and then rendered using a skin material modeled with subsurface
scattering [38], [39]. Because the craniofacial surfaces are not used for quantitiative purposes, the smoothing step does not introduce any error in calculations. If the quantitative use of craniofacial surfaces is desired, the post-processing step can easily be omitted.

4.3 Execution time

Execution time results were computed from trials ran on each of the 14 samples. GPGPU trials were executed on a GeForce 9800 GT with 1 GB of graphics memory while CPU trials were performed by an Intel Pentium 4 3.0 GHz processor (model 631) with 2 GB of main memory. The trials were ran on the Linux operating system version 2.6.28-11 with a display driver that supports CUDA version 2.2.

GPGPU times were obtained from CUDA event timers and CPU times were obtained from linux system calls. Both timers provided microsecond-granularity readings that were converted to milliseconds (ms) for display purposes. Table 4.1 shows an overview comparison of the times obtained from the GPGPU trials versus CPU times. These execution times are collected for each of the 14 using the default gaussian filter width of 2. Parallel speedup factors are given as the ratio of the CPU time to the GPU time. These speedup metrics are defined as the ratio of the GPU implementation of the algorithms given in chapter 3 to the same algorithms implemented on the CPU. While care was taken during the CPU implementation of the algorithms,
Figure 4.1. Airway segmentation surfaces (samples 1-9)
Figure 4.2. Airway segmentation surfaces (samples 9-14)
Figure 4.3. Craniofacial surface renderings (samples 1-6)
Figure 4.4. Craniofacial surface renderings (samples 6-12)
Figure 4.5. Craniofacial surface renderings (samples 13-14)
it is important to note that they may not be the most efficient implementations of these algorithms for the CPU.

The individual algorithm execution times for the GPGPU are given in table 4.2 and do not include the time for input-output (IO) operations such as the allocation and transfer of data down the PCI-express bus to the graphics chip. The total values, however do account for the overhead incurred from IO. Because of this, the values displayed for the total shown do not reflect the sum of the individual times, but the time taken for the entire process to compute from the end of the volume loading to the completion of the airway surfacing. Table 4.3 contains the individual algorithm times for the CPU. The individual speedup factors for each algorithm are given in table 4.4.

Table 4.1. GPGPU vs CPU timing totals

<table>
<thead>
<tr>
<th></th>
<th>GPU Time (ms)</th>
<th>CPU Time (ms)</th>
<th>Speedup Factor</th>
<th>Volume Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>384.74</td>
<td>11015.20</td>
<td>28.6</td>
<td>400x400x321</td>
</tr>
<tr>
<td>2</td>
<td>683.94</td>
<td>15434.00</td>
<td>22.6</td>
<td>534x534x429</td>
</tr>
<tr>
<td>3</td>
<td>381.99</td>
<td>10576.80</td>
<td>27.7</td>
<td>400x400x321</td>
</tr>
<tr>
<td>4</td>
<td>378.40</td>
<td>10559.20</td>
<td>27.9</td>
<td>400x400x321</td>
</tr>
<tr>
<td>5</td>
<td>379.50</td>
<td>10536.40</td>
<td>27.8</td>
<td>400x400x324</td>
</tr>
<tr>
<td>6</td>
<td>378.56</td>
<td>10563.20</td>
<td>27.9</td>
<td>400x400x321</td>
</tr>
<tr>
<td>7</td>
<td>380.70</td>
<td>10892.40</td>
<td>28.6</td>
<td>400x400x321</td>
</tr>
<tr>
<td>8</td>
<td>382.39</td>
<td>10545.60</td>
<td>27.6</td>
<td>400x400x323</td>
</tr>
<tr>
<td>9</td>
<td>378.75</td>
<td>9980.92</td>
<td>26.4</td>
<td>400x400x321</td>
</tr>
<tr>
<td>10</td>
<td>379.09</td>
<td>10875.40</td>
<td>28.7</td>
<td>400x400x323</td>
</tr>
<tr>
<td>11</td>
<td>379.32</td>
<td>10244.50</td>
<td>27.0</td>
<td>400x400x321</td>
</tr>
<tr>
<td>12</td>
<td>379.38</td>
<td>10211.90</td>
<td>26.9</td>
<td>400x400x323</td>
</tr>
<tr>
<td>13</td>
<td>380.34</td>
<td>11171.40</td>
<td>29.4</td>
<td>400x400x323</td>
</tr>
<tr>
<td>14</td>
<td>386.04</td>
<td>12754.60</td>
<td>33.0</td>
<td>400x400x323</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>11097.25</td>
<td>27.9</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 4.2. GPGPU individual times for Geforce 9800 GT (ms)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>58.45</td>
<td>146.23</td>
<td>0.72</td>
<td>24.22</td>
<td>55.43</td>
<td>36.48</td>
</tr>
<tr>
<td>2</td>
<td>123.30</td>
<td>268.18</td>
<td>0.72</td>
<td>35.34</td>
<td>108.3</td>
<td>35.66</td>
</tr>
<tr>
<td>3</td>
<td>58.44</td>
<td>148.03</td>
<td>0.69</td>
<td>24.08</td>
<td>56.42</td>
<td>36.04</td>
</tr>
<tr>
<td>4</td>
<td>57.97</td>
<td>148.05</td>
<td>0.78</td>
<td>22.81</td>
<td>53.68</td>
<td>36.05</td>
</tr>
<tr>
<td>5</td>
<td>58.70</td>
<td>146.37</td>
<td>0.67</td>
<td>23.97</td>
<td>54.31</td>
<td>35.89</td>
</tr>
<tr>
<td>6</td>
<td>57.95</td>
<td>148.91</td>
<td>0.68</td>
<td>22.40</td>
<td>54.29</td>
<td>35.59</td>
</tr>
<tr>
<td>7</td>
<td>58.45</td>
<td>147.54</td>
<td>0.66</td>
<td>23.43</td>
<td>56.11</td>
<td>35.91</td>
</tr>
<tr>
<td>8</td>
<td>58.09</td>
<td>148.46</td>
<td>0.70</td>
<td>23.58</td>
<td>57.04</td>
<td>35.42</td>
</tr>
<tr>
<td>9</td>
<td>57.97</td>
<td>148.10</td>
<td>0.65</td>
<td>22.87</td>
<td>54.59</td>
<td>35.87</td>
</tr>
<tr>
<td>10</td>
<td>58.56</td>
<td>144.55</td>
<td>0.64</td>
<td>24.23</td>
<td>56.87</td>
<td>35.40</td>
</tr>
<tr>
<td>11</td>
<td>58.44</td>
<td>150.13</td>
<td>0.68</td>
<td>22.44</td>
<td>53.63</td>
<td>35.46</td>
</tr>
<tr>
<td>12</td>
<td>58.54</td>
<td>150.57</td>
<td>0.75</td>
<td>22.28</td>
<td>51.75</td>
<td>35.42</td>
</tr>
<tr>
<td>13</td>
<td>58.10</td>
<td>149.94</td>
<td>0.72</td>
<td>22.16</td>
<td>53.68</td>
<td>35.78</td>
</tr>
<tr>
<td>14</td>
<td>59.14</td>
<td>149.62</td>
<td>0.76</td>
<td>22.81</td>
<td>57.47</td>
<td>37.33</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>63.01</td>
<td>156.76</td>
<td>0.70</td>
<td>24.04</td>
<td>58.83</td>
</tr>
</tbody>
</table>

Table 4.3. Individual times for CPU (ms)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1081.39</td>
<td>438.25</td>
<td>14.35</td>
<td>5472.21</td>
<td>470.48</td>
<td>3125.20</td>
</tr>
<tr>
<td>2</td>
<td>2570.52</td>
<td>1158.15</td>
<td>21.47</td>
<td>7025.46</td>
<td>983.91</td>
<td>2993.55</td>
</tr>
<tr>
<td>3</td>
<td>1079.52</td>
<td>438.62</td>
<td>14.49</td>
<td>5494.14</td>
<td>452.97</td>
<td>2673.71</td>
</tr>
<tr>
<td>4</td>
<td>1098.22</td>
<td>508.14</td>
<td>15.10</td>
<td>5303.90</td>
<td>433.71</td>
<td>2778.94</td>
</tr>
<tr>
<td>5</td>
<td>1128.20</td>
<td>450.82</td>
<td>14.97</td>
<td>5498.26</td>
<td>434.93</td>
<td>2559.81</td>
</tr>
<tr>
<td>6</td>
<td>1080.02</td>
<td>471.34</td>
<td>13.76</td>
<td>5266.77</td>
<td>420.86</td>
<td>2884.54</td>
</tr>
<tr>
<td>7</td>
<td>1078.28</td>
<td>448.99</td>
<td>13.84</td>
<td>5400.03</td>
<td>434.01</td>
<td>3079.90</td>
</tr>
<tr>
<td>8</td>
<td>1084.67</td>
<td>464.00</td>
<td>13.65</td>
<td>5584.57</td>
<td>435.08</td>
<td>2535.52</td>
</tr>
<tr>
<td>9</td>
<td>1106.11</td>
<td>475.46</td>
<td>14.88</td>
<td>5278.88</td>
<td>420.99</td>
<td>2263.55</td>
</tr>
<tr>
<td>10</td>
<td>1135.07</td>
<td>429.45</td>
<td>13.80</td>
<td>5683.59</td>
<td>438.11</td>
<td>2750.87</td>
</tr>
<tr>
<td>11</td>
<td>1087.16</td>
<td>489.48</td>
<td>14.72</td>
<td>5181.27</td>
<td>428.53</td>
<td>2623.84</td>
</tr>
<tr>
<td>12</td>
<td>1089.12</td>
<td>546.52</td>
<td>15.16</td>
<td>5183.88</td>
<td>439.50</td>
<td>2485.51</td>
</tr>
<tr>
<td>13</td>
<td>1098.94</td>
<td>496.24</td>
<td>14.87</td>
<td>5227.25</td>
<td>422.82</td>
<td>3427.17</td>
</tr>
<tr>
<td>14</td>
<td>1093.86</td>
<td>493.61</td>
<td>15.10</td>
<td>5277.89</td>
<td>438.13</td>
<td>5007.12</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>1200.79</td>
<td>522.08</td>
<td>15.01</td>
<td>5491.29</td>
<td>475.29</td>
</tr>
<tr>
<td>----</td>
<td>----------</td>
<td>-----</td>
<td>-----------</td>
<td>-------------</td>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td>1</td>
<td>18.5</td>
<td>3.0</td>
<td>19.9</td>
<td>225.9</td>
<td>8.5</td>
<td>85.7</td>
</tr>
<tr>
<td>2</td>
<td>20.8</td>
<td>4.3</td>
<td>30.0</td>
<td>198.8</td>
<td>9.1</td>
<td>83.9</td>
</tr>
<tr>
<td>3</td>
<td>18.5</td>
<td>3.0</td>
<td>20.9</td>
<td>228.2</td>
<td>8.0</td>
<td>74.2</td>
</tr>
<tr>
<td>4</td>
<td>18.9</td>
<td>3.4</td>
<td>19.5</td>
<td>232.5</td>
<td>8.1</td>
<td>77.1</td>
</tr>
<tr>
<td>5</td>
<td>19.2</td>
<td>3.1</td>
<td>22.4</td>
<td>229.3</td>
<td>8.0</td>
<td>71.3</td>
</tr>
<tr>
<td>6</td>
<td>18.6</td>
<td>3.2</td>
<td>20.3</td>
<td>235.2</td>
<td>7.8</td>
<td>81.0</td>
</tr>
<tr>
<td>7</td>
<td>18.4</td>
<td>3.0</td>
<td>21.1</td>
<td>230.5</td>
<td>7.7</td>
<td>85.8</td>
</tr>
<tr>
<td>8</td>
<td>18.7</td>
<td>3.1</td>
<td>19.4</td>
<td>236.8</td>
<td>7.6</td>
<td>71.6</td>
</tr>
<tr>
<td>9</td>
<td>19.1</td>
<td>3.2</td>
<td>22.8</td>
<td>230.9</td>
<td>7.7</td>
<td>63.1</td>
</tr>
<tr>
<td>10</td>
<td>19.4</td>
<td>3.0</td>
<td>21.7</td>
<td>234.6</td>
<td>7.7</td>
<td>77.7</td>
</tr>
<tr>
<td>11</td>
<td>18.6</td>
<td>3.3</td>
<td>21.7</td>
<td>230.9</td>
<td>8.0</td>
<td>74.0</td>
</tr>
<tr>
<td>12</td>
<td>18.6</td>
<td>3.6</td>
<td>20.1</td>
<td>232.7</td>
<td>8.5</td>
<td>70.2</td>
</tr>
<tr>
<td>13</td>
<td>18.9</td>
<td>3.3</td>
<td>20.6</td>
<td>235.9</td>
<td>7.9</td>
<td>95.8</td>
</tr>
<tr>
<td>14</td>
<td>18.5</td>
<td>3.3</td>
<td>19.7</td>
<td>231.4</td>
<td>7.6</td>
<td>134.1</td>
</tr>
<tr>
<td>Average</td>
<td>18.9</td>
<td>3.3</td>
<td>21.4</td>
<td>229.5</td>
<td>8.0</td>
<td>81.8</td>
</tr>
</tbody>
</table>
Using the CUDA visual profiler, each kernel invocation is analyzed for time spent executing on the GPU. These times are given in microseconds and are possibly more accurate than the times retrieved by the software at run time. Figure 4.6 contains a plot of GPU execution time percentages generated using the profiler output. The chart shows which methods the GPU spent most of its time executing.

Figure 4.6. Plot of GPU time percentages for each method

The 3D gaussian algorithm used is the direct convolution approach as discussed in section 3.2.1 for use with small filter widths. Results for the separable 3D gaussian are provided in table 4.5 which shows how the gaussian filter width affects the computation times on the GPU and CPU, respectively. The 1D gaussian method shown here refers to the gaussian postprocessing step of the airway detection algorithm outlined in section 3.2.3. Likewise, the three airway surfacing methods are components of the marching cubes algorithm used to surface the airway as discussed in section 3.2.5.
Table 4.5. Gaussian filter times for different filter widths (times in ms)

<table>
<thead>
<tr>
<th></th>
<th>GPU(5)</th>
<th>CPU(5)</th>
<th>Speedup(5)</th>
<th>GPU(10)</th>
<th>CPU(10)</th>
<th>Speedup(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>545.46</td>
<td>6066.42</td>
<td>11.1</td>
<td>952.48</td>
<td>9113.34</td>
<td>9.6</td>
</tr>
<tr>
<td>2</td>
<td>1124.53</td>
<td>14434.80</td>
<td>12.8</td>
<td>1976.35</td>
<td>21908.80</td>
<td>11.1</td>
</tr>
<tr>
<td>3</td>
<td>546.28</td>
<td>6311.92</td>
<td>11.6</td>
<td>953.28</td>
<td>9109.53</td>
<td>9.6</td>
</tr>
<tr>
<td>4</td>
<td>546.92</td>
<td>6035.77</td>
<td>11.0</td>
<td>953.69</td>
<td>9157.63</td>
<td>9.6</td>
</tr>
<tr>
<td>5</td>
<td>549.07</td>
<td>6094.40</td>
<td>11.1</td>
<td>962.64</td>
<td>9187.74</td>
<td>9.5</td>
</tr>
<tr>
<td>6</td>
<td>545.93</td>
<td>6039.98</td>
<td>11.1</td>
<td>953.07</td>
<td>9220.57</td>
<td>9.7</td>
</tr>
<tr>
<td>7</td>
<td>545.96</td>
<td>6091.99</td>
<td>11.2</td>
<td>952.00</td>
<td>9129.75</td>
<td>9.6</td>
</tr>
<tr>
<td>8</td>
<td>548.86</td>
<td>6149.16</td>
<td>11.2</td>
<td>959.15</td>
<td>9158.95</td>
<td>9.5</td>
</tr>
<tr>
<td>9</td>
<td>546.26</td>
<td>6061.90</td>
<td>11.1</td>
<td>952.01</td>
<td>9188.16</td>
<td>9.7</td>
</tr>
<tr>
<td>10</td>
<td>549.08</td>
<td>6071.81</td>
<td>11.1</td>
<td>958.37</td>
<td>9170.58</td>
<td>9.6</td>
</tr>
<tr>
<td>11</td>
<td>545.87</td>
<td>6088.39</td>
<td>11.2</td>
<td>951.64</td>
<td>9104.62</td>
<td>9.6</td>
</tr>
<tr>
<td>12</td>
<td>548.97</td>
<td>6093.19</td>
<td>11.1</td>
<td>958.07</td>
<td>9189.09</td>
<td>9.6</td>
</tr>
<tr>
<td>13</td>
<td>548.70</td>
<td>6132.15</td>
<td>11.2</td>
<td>959.29</td>
<td>9138.73</td>
<td>9.5</td>
</tr>
<tr>
<td>14</td>
<td>549.10</td>
<td>6139.61</td>
<td>11.2</td>
<td>957.84</td>
<td>9231.66</td>
<td>9.6</td>
</tr>
<tr>
<td>Average</td>
<td>588.64</td>
<td>6700.82</td>
<td>11.3</td>
<td>1028.56</td>
<td>10072.08</td>
<td>9.7</td>
</tr>
</tbody>
</table>

4.3.1 Occupancy

Individual multiprocessor occupancy statistics were collected for each method executed on the GPU. These statistics were calculated by the CUDA visual profiler. As mentioned in 1.1.2 the occupancy for a kernel, is the ratio of active warps to total warps available on a multiprocessor. These statistics are highly dependent on block size and are invariant over the sample space. The occupancies are given in table 4.6.
Table 4.6. Multiprocessor occupancies for GPU kernels

<table>
<thead>
<tr>
<th>Method</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_gaussian1d</td>
<td>1</td>
</tr>
<tr>
<td>d_airway_surface_compact</td>
<td>1</td>
</tr>
<tr>
<td>d_gaussian3d</td>
<td>0.67</td>
</tr>
<tr>
<td>d_airway_segment</td>
<td>0.67</td>
</tr>
<tr>
<td>d_trace_cranial</td>
<td>0.67</td>
</tr>
<tr>
<td>d_airway_surface_classify</td>
<td>0.5</td>
</tr>
<tr>
<td>d_pca</td>
<td>0.33</td>
</tr>
<tr>
<td>d_find_airway_center</td>
<td>0.33</td>
</tr>
<tr>
<td>d_airway_surface_triangular</td>
<td>0.13</td>
</tr>
</tbody>
</table>
Chapter 5

Discussion

5.1 Segmentation

The segmentation results and their corresponding surfaces experience false positives only during cases where the size is overestimated during the airway detection step. These false positives in the detection occur in the slice-boundaries of the detection algorithm and can best be seen in samples 3 and 8 from figure 4.1. In many cases, a simple background segmentation can eliminate these artifacts during the airway segmentation step. For the slices above the soft palate the algorithm occasionally includes the empty space in the mouth below the tongue. In these cases, the extra volume appears as a separate body within the segmentation surface result. This scenario occurs during the airway segmentation of sample 13.

Because the samples are ranked from highest to lowest PSNR, figures 4.1 and 4.2 show how the algorithm responds to degradation in the cone-beam sets due to noise. Visually, only samples 13 and 14 suffer from the heavy noise degradation. The noise in these samples clearly affects the segmentation and propagates through the marching cubes resurfacing. To show the benefit of the gaussian 3D, samples 13 and 14 were recomputed using a larger gaussian filter width. The results of the recomputation is given in figure 5.1 which shows an improvement over the original.
Certain features available to GPU hardware that are exposed by CUDA are not easily reproducible during CPU implementation. Such features include hardware filtering modes for texture access and normalization in addressing and integer values. For these reasons, CPU segmentation results and GPU results were similar, but not identical. Thus, a strict vertex-for-vertex comparison of the CPU and GPU generated meshes is not adequate for verification reasons. However, a visual comparison shows that CPU and GPU results are similar.

Regarding the scenario discussed in section 3.2.2 for the PCA computation, when the eigenvalues are similar, the eigenvectors have little meaning and cannot be relied on for airway detection. This situation did occur during the some of the slices of the segmentation for a subset of the samples. However, the airway detection and segmentation did not fail for these cases due to incorporation of alternative solutions presented in section 3.2.3 and the post-processing gaussian filter step.
The airway segmentation results were not as sensitive to noise as the craniofacial traces. This can be seen in the traces for the higher samples depicted in figures 4.4 and 4.5. Among the reasons for the quality loss in these facial traces, are metal artifacts due to X-ray scatter from various dental implants. Another possible cause of degradation is a reduced number of projections used to reconstruct the CT volume. These numbers were not available for the purposes of this study. Unlike the airway surfaces, the craniofacial surfaces presented in section 4.2 were post-processed using a mesh smoothing algorithm implemented outside of the experimental software. The images in figure 3.18 show the unprocessed output surfaces from the craniofacial trace.

5.2 Performance

The segmentation approach presented in chapter 3 when implemented on the GPU performed faster than on the CPU. On average, the GPU performed the segmentation nearly 28 times faster than the CPU. As the volume size increased, the GPU outperforms the CPU as well, but by a smaller factor. Similarly, as the gaussian filter width increases, the speedup factor seems to decrease. While the direct cause of this is not known, one possible explanation results from the consideration of CPU cache memory. Cache memory allows for increased performance on the CPU over longer periods of execution. Modern BIOSes support mechanisms that modulate voltage supplies to CPUs in direct response to heavy load [40], thus affecting its long-term performance during a segmentation.
The occupancy ratios given in table 4.6 show the ratio of active to inactive warps for a given multiprocessor during GPU execution. Reduced values indicate a serialization or delay in thread-block execution while a value of 1.0 indicates complete utilization of the GPU during the entire kernel execution. Among the individual GPU kernels implemented for this study, the 1D post-processing Gaussian filter, which functions as a subroutine of the airway detection ran with the highest occupancy of 1.0. This was followed by the 3D gaussian, airway segmentation, and cranial trace kernels which had occupancies of 0.67. The per-slice kernels that compute the PCA and detect the airway location had values of 0.33. The multiple stages of the marching cubes algorithm had varying occupancies ranging from 1.0 to 0.13. These values indicate that some of the algorithms implemented are more inherently parallel than the others.

Figure 4.6 shows that the PCA computation is the most expensive procedure executed on the GPU, taking nearly 50% of the total GPU computation time. This is likely due to the per-slice approach used to implement the PCA in parallel. Unlike the Gaussian and airway surfacing kernels, the PCA kernel computes over each slice inside of its own thread, resulting in a lower occupancy for each GPU multiprocessor. On average, the per-slice PCA step took 156 ms to complete on a CT volume. In an attempt to increase performance, the PCA was implemented using a parallel reduction for prefix sums similar to the method used by the marching cubes GPU implementation. However, the overhead involved in the I/O transfers of the covariance
data between the host and the device along with the limited device memory available resulted in the failure of the parallel approach to outperform the original. However, it is interesting to note that while the time performance of the parallel reduction was low, the GPU occupancy was increased. This is an indication that increased GPU occupancy does not always imply a performance increase when memory is limited.

Additional factors introduced variability in the execution times for both the GPGPU and CPU implementations. Apart from the obvious factors such as volume size and clock speed, CPU cache size and available memory played a large role in the algorithm performance. Due to the large memory requirement of the marching cubes algorithm, implementation of the airway surfacing during a single kernel invocation requires a minimum of 1 GB of available memory. If such memory is not available, the algorithm execution must be divided into multiple kernel launches. Because each launch will be followed by a transfer of the results back to host memory, overhead from these I/O transfers will be incurred resulting in a performance penalty. Another significant source of performance variability among the individual GPU algorithms was the complexity of the algorithms themselves. This was due to the per-kernel register count limitation discussed in section 1.1.2. This limitation forced the introduction of additional branches in the separable 3D gaussian kernel which was used only for large filter widths. The limitation also required the use of smaller block sizes for some kernels which may have affected execution performance as well. These register count limitations were not present on the CPU implementation.
On the topic of scalability, this author noticed a significant drop in performance when the algorithm was ran on lower-end GPUs. The most apparent explanation for this is the difference in number of scalar multiprocessor cores between lower end chips such as the NVIDIA 9400M which has 16 cores and the NVIDIA 9800 GT which has 112 cores. Early estimates show that the algorithm scales very well which the SM core counts. This is likely due to the CUDA architecture which dynamically optimizes each CUDA grid to execute optimally on each architecture [11]. What is not as easily identifiable, is how the algorithm scales with memory sizes. This is due to the rate at which GPU memory is increasing in comparison to processing power (SM counts). While a $512^3$ volume can be processed using the methods of this study on a GPU with 1 GB of graphics memory, a $1024^3$ volume would barely fit on the same GPU. Additionally, access to more than 4 GB of graphics memory requires a 64-bit architecture on the GPU. Due to the complex nature of the GPU architecture, and limited memory, models of such scalable performance may not be accurate.
Chapter 6

Conclusions and Future Work

The methods presented in this work successfully accomplished the goal of the study: to detect and segment the upper airway from cone-beam CT volumes in a fully automatic algorithm implemented on the GPU. Thus, no user intervention was required to produce the segmentation result. Furthermore, all of the GPGPU methods presented significantly outperformed their CPU counterparts with sub-second latency resulting in a real-time airway segmentation solution that is feasible for interactive applications. Lastly, the initial hypothesis was proven in that GPU methods can reliably be used to accelerate CT volume analysis and segmentation with results suitable for OSA diagnostics. Work is planned for the quantification of airway data, along with collection of landmarks, linear measurements, and statistics pertinent to OSA. Additionally, further validation is planned to compare the automatic segmentation results to those of a manual segmentation method. The signal processing and statistical aspect of the study proved invaluable as it introduced techniques that were integral to the successful detection and segmentation of the airway. The experimental craniofacial trace solution also proved useful as an inexpensive method for surface extraction from CT without the reliance on marching cubes. The implementation of this segmentation approach on a commodity graphics card makes it a viable solution for end-user desktop computing. The polygonal airway surface meshes resulting from the
computation are suitable for importation into visual or analytical software. Specifically, the use of finite element methods (FEM) in morphology [41] can be directly applied to segmented airway surfaces to analyze deformations. Such analysis will aid diagnostic work in the field of sleep apnea research. Additional work is also planned for the development of input filters for medical imaging formats such as dicom and vff to aid in software flexibility.

As the state-of-the-art advances, future research is planned for implementation of segmentation algorithms using new technologies such as OpenCL [42]. OpenCL provides additional flexibility as currently CUDA is only available on NVIDIA graphics hardware. Additionally, techniques such as the mesh smoothing applied to the craniofacial renderings in section 4.2 should be implemented on the GPU along with additional preprocessing filters to combat metal artifacts in the cone-beam volumes [43]. As an extra analytical step, a 3D PCA can be computed on the airway volume itself to add meaning and alignment to the data for comparison with other sample sets. For the 2D PCA, a parallel reduction method will be revisited to further optimize the GPU implementation. Ideally in practice, the CBCT volume would be obtained directly from the output of a GPU process used to reconstruct the volume using the aforementioned backprojection techniques as discussed in section 2.1. This would eliminate the necessity of storing and loading CT image volumes within individual image slices on the disk. Furthermore, it would eliminate any compression artifacts due to JPEG compression introduced during the same process. Because of the inher-
ently parallel nature of the GVF approach presented in [20], it would be interesting to explore such an implementation on the GPU and compare it with other airway segmentation techniques such as the thresholding mechanism used in this work.
Bibliography


