2018-01-04

GPU-based Compressive Volume Rendering

Rabbani, Md Reza

UNIVERSITY OF CALGARY

GPU-based Compressive Volume Rendering

by

Md. Reza Rabbani

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

January, 2018

Abstract

There is an increasing demand in the scientific visualization community for high quality real time interactive volume renderers; but the goal of high quality in volume rendering degrades the performance of the volume renderer. The current advancements in graphics hardware has resulted in the adoptation of the GPU as a solution for the degradation issue in a volume renderer. However there is a caveat, with the use of the GPU as a solution; as the GPUs memory size and long data transfer times between CPU and GPU limit the performance of the GPU based volume renderer.

The GPU based volume renderer performance issue can be resolved by rendering a subset of the pixels. By reducing the volume of data the computational costs are reduced. Then using a GPU based conjugate gradient solver we can reconstruct the full image which has the same quality as the original image. This dissertation will show how using GPUs and compressive rendering will optimize the performance of a volume renderer.

Acknowledgements

After an intensive period of months, I am finally writing this note of thanks as a finishing touch to my thesis. It has been a time of intense learning for me, not only in the computer science field but also on a personal level. Completing my graduate degree by writing this thesis has had a big impact on me. I would like to reflect on the people who have supported and helped me so much throughout this endeavor.

I would never have been able to finish my thesis without the guidance of my supervisor Dr. Usman R. Alim. I would like to express my deepest gratitude to Dr. Usman R. Alim for his endless support, caring, and patience through the learning process of my master's studies. I would also like to thank Dr. Leonard Manzara for giving me the opportunity for being a teaching assistant in multiple courses which enlightened my knowledge.

Thanks to all the group members of Visualization and Graphics Group (VISAGG) for helping me throughout my Masters degree. I really enjoyed every Wednesday wings night at the Den with the VISAGG group. I am also grateful to Haysn Hornbeck for his valuable comments and suggestion in this thesis work.

I would like to thank my examination committee members, Dr Mea Wang and Dr Wenyuan Liao for being my committee members and providing valuable remarks and advice.

Table of Contents

Abstract . . . ii

Acknowledgements . . . iii

Table of Contents . . . iv

List of Tables . . . vi

List of Figures . . . vii

1 Introduction . . . 1

1.1 Background . . . 1

1.2 Rise of the GPU . . . 2

1.3 Contributions . . . 3

1.4 Organization . . . 4

2 Related Work . . . 6

2.1 Volume Rendering . . . 6

2.1.1 Volume Data . . . 7

2.1.2 Ray Casting . . . 7

2.1.3 Volume Visualization Tools . . . 10

2.2 Literature Review . . . 10

2.2.1 GPU based volume rendering . . . 10

2.2.2 Compression domain volume rendering . . . 12

2.3 CPUs vs GPUs . . . 13

2.4 Parallel Programming Model . . . 15

2.5 CUDA Programming Model . . . 16

2.5.1 CUDA Threads . . . 17

2.5.2 CUDA Blocks . . . 18

2.5.3 CUDA Memory . . . 19

3 Compressive Volume Rendering . . . 22

3.1 Overview . . . 22

3.2 Problem Definition . . . 23

3.3 Recovery of Missing Pixels using CUDA . . . 26

3.3.1 Initialization . . . 27

3.3.2 Reconstruction via a Global Solver . . . 27

3.3.3 Convolution . . . 29

3.3.4 Vector multiplication . . . 29

3.3.5 Vector Addition/Subtraction . . . 30

3.3.6 Vector Dot Product . . . 30

4 CGS based Compressive Volume Rendering via a Local Solver . . . 31

4.1 Optimizing CUDA application . . . 31

4.2 Local Conjugate Gradient Solver . . . 32

4.2.1 Background . . . 33

4.2.2 Padding with Ghost Elements . . . 33

4.2.3 Thread and Block Configuration . . . 34

4.2.4 Using Shared Memory . . . 35

4.2.5 Shared memory initialization . . . 35

4.2.6 Convolution 36
4.2.7 Vector Dot Product 38
4.3 Adaptive Sampling 39
5 Results and Discussions 41
5.1 High quality rendering 41
5.1.1 Step size 41
5.1.2 Interpolation method 43
5.1.3 Shading model 43
5.1.4 Super sampling 45
5.2 Quality vs Performance 45
5.3 Compressive Rendering 46
5.3.1 Global solver 47
5.3.2 Local solver 48
5.3.3 Quality of local solver 58
5.3.4 Comparison of Global and Local solvers 60
5.4 Adaptive Sampling 61
5.5 CUDA profiling 63
5.5.1 Local Solver 63
5.5.2 Adaptive Sampling 65
6 Conclusion 69
6.1 Summary 69
6.2 Contributions 69
6.3 Future Work 70
Bibliography 72

List of Tables

2.1 Overview of different types of GPU memory 20

List of Figures and Illustrations

1.1 A fixed function NVIDIA graphics pipeline [1] 2

2.1 Ray casting through a volume [2] 8
2.2 Stages of volume rendering 9
2.3 Higher level comparison of CPU and GPU [1] 14
2.4 CUDA Execution Model 16
2.5 A Von-Neumann Processor 17
2.6 Arrays of parallel threads 18
2.7 Thread Blocks 19
2.8 CUDA memory model [3] 19

3.1 Binary mask and Rendered image with missing pixel using the mask 27

4.1 Padded Mask 34
4.2 Parallel Reduction 38
4.3 Flow chart for adaptive sampling strategy 39

5.1 Image quality for different step sizes 42
5.2 Comparison of images generated using three different interpolation methods 43
5.3 Comparison of images while turning on light vs without lighting 44
5.4 Super sampling vs regular sampling 45
5.5 Changes of FPS for different interpolation methods 46
5.6 Reconstruction time of global solver for various missing pixel 47
5.7 Reconstruction time of local solver for various missing pixel 48
5.8 Timing diagram for tri-linear interpolation without shading 50
5.9 Timing diagram for tri-linear interpolation with shading 51
5.10 Timing diagram for tri-linear interpolation with super-sampling 52
5.11 Timing diagram for tri-linear interpolation with super-sampling and shading 53
5.12 Timing diagram for tri-cubic interpolation without shading 54
5.13 Timing diagram for tri-cubic interpolation with shading 55
5.14 Timing diagram for tri-cubic interpolation with super-sampling 56
5.15 Timing diagram for tri-cubic interpolation with super-sampling and shading 57
5.16 PSNR for different missing pixels 58
5.17 PSNR for different missing pixels 59
5.18 PSNR for different missing pixels for iso-surface rendering 59
5.19 Comparison between global and local solvers for the foot image shown in Fig. 5.3 (top-left). 60
5.20 Timing diagram for adaptive sampling with tri-linear interpolation method 62
5.21 Timing diagram for adaptive sampling with tri-cubic interpolation method 62
5.22 CUDA visual profiler for local solver 63
5.23 Volume renderer kernel description 64
5.24 Reconstruction kernel description 64
5.25 CUDA visual profiler for adaptive sampling 65

5.26 The profiler for the kernel that casted rays for padded region 66
5.27 The profiler for the kernel that casted 12.5% rays for each block 66
5.28 The profiler for the kernel that casted additional rays for each block 67
5.29 The profiler for the conjugate gradient solver kernel 67

Chapter 1

Introduction

This thesis is intended to explore the optimization of the performance of a GPU based real time interactive volume renderer. The context and challenges of this research will be discussed and their solutions investigated. The inspiration of this thesis was based on the work done by Liu and Alim [4].

1.1 Background

According to Wikipedia [5], volume rendering refers to a set of techniques used to display a 2D projection of a 3D scalar field. In general, volume rendering involves the following steps; creating a RGBA pixel value from the volume data, constructing a continuous function from the discretely sampled volume data and finally projecting it onto a 2D viewing plane. The biggest advantage of volume rendering is that it never has interior information degradation., so that it enables the 3D data set to be viewed as a whole.

The traditional approach to CPU based volume rendering consists of pixel by pixel operations. These operations cast rays for each pixel and are done sequentially; therefore the more pixels there are the more rendering time it takes. Another challenge is accommodating regular volume renderers with large screen displays. Some other considerations such as sophisticated illumination techniques and high quality rendering makes the process more costly and it also causes problems of user interactivity. This is why it is important to optimize the performance of volume renderers.

Liu and Alim et al. [4] proposed that compressive volume rendering would save computational costs in traditional volume rendering. Their work explored four different image techniques and the smoothness-based method outperformed all others. Their studies show that a high recovery of a rendered image can be possible with as little as 20% of the pixels.

Although the work by Liu and Alim et al. [4] allows image rendering with as little as 20% of the pixels the performance of the render is severely hampered. The large time allowance for recovering missing pixels make it infeasible to be used in interactive volume rendering. This thesis is intended to bridge the gap between performance and interactivity by focusing on the GPU's parallel architecture to accelerate the reconstruction process.

1.2 Rise of the GPU

Since the 1980's scientists have worked on improving the performance of the CPU (Central Processing Unit). They approached this improvement by increasing the processor's clock speed; and after 30 years the clock speeds increased almost a thousand fold. Manufacturers then looked for other alternatives to increase the performance of the CPU and started producing multi-core CPUs; which increased the computing performance.

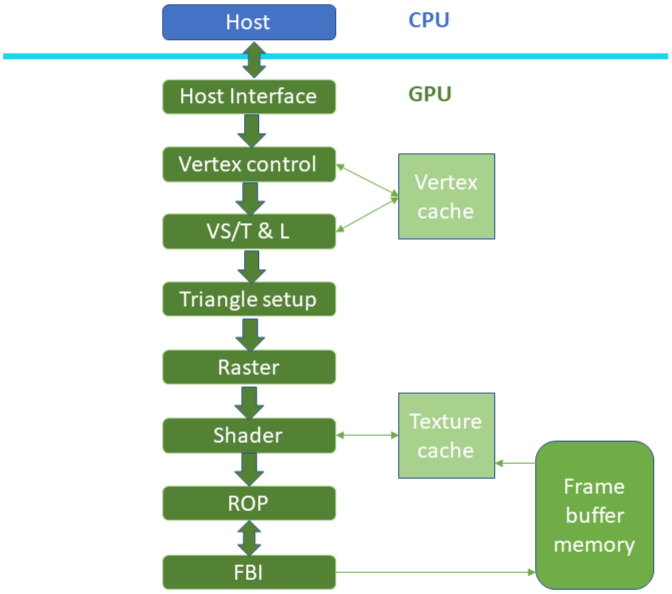


Figure 1.1: A fixed function NVIDIA graphics pipeline [1]

The remarkable advancement of graphics hardware is driven by the need for improved performance in real time high quality graphics. The graphics hardware in the 1980s to 1990s are fixed-function pipelines as shown in figure 1.1. The GPUs in this time period are primarily used for graphical applications; at the same time developers are creating more sophisticated programs and requesting for more features to fit into built-in functions of the GPU. Since the GPUs are configurable but not programmable the next step is to make these graphics pipeline stages into programmable processors. With further development, the GPUs are found to outperform the computational speed compared to multi-core CPUs. NVIDIA then realizes that "parallelism" is a necessity for faster performance and in 2001 with its GeForce 3 series of GPUs [6] the concept of GPGPU(General Purpose Graphics Processing Unit) is introduced to the world.

Volume rendering consists of pixel by pixel operations where individual pixel operations are independent. This application is suitable for parallel execution and with a good implementation on the GPGPU it can achieve significant speedup over sequential execution. In this thesis NVIDIA's CUDA (Compute Unified Device Architecture) is used; CUDA is one of the parallel programming models that supports the hardware in facilitating parallel implementation

1.3 Contributions

This thesis is inspired by the work done by Liu and Alim et al. [4]; they have investigated four different methods of recovering missing pixels. The first technique is based on the theory of compressed sensing and leverages the sparsity of the image gradient in the Fourier domain. The second methodology is investigating the sparse representation of the volume rendered images via dictionary learning. The final two techniques looks at exploiting the smoothness properties of the rendered image. The third process recovers the missing pixels via a total minimization procedure while the final method incorporates a smoothness process prior to a variational reconstruction framework employing interpolating cubic B-splines.

The *Smoothness Splines* recovery method outperformed all other recovery methods but suffered from performance issues which manifested as long reconstruction times which are unacceptable for real time applications. The main focus of this thesis is to design a conjugate gradient solver for real time applications such as volume rendering. In order to achieve this, parallel computing will be exploited in the GPU. This thesis details two different proposals for the conjugate gradient solver.

1. Global Solver: The conjugate gradient solver is implemented as an iterative method that is applicable to sparse systems that are too cumbersome to be handled. The CPU based conjugate gradient solver takes a significant amount of time and it cannot be used in real time applications. The use of a GPU is one of the solutions to deal with the processing time issues of a solver; by running the entire gradient conjugate solver for the entire image this doesn't guarantee optimal performance that can be used in real time direct volume rendering. In this proposal, there is an inordinate number of times the solver accesses global memory which results in longer processing times. Although the timing is significantly faster than the work done by Liu and Alim et al. [4]. it is not feasible for real time applications.

2. Local Solver: In this technique the processing is optimized as the data is partitioned into blocks, padded with known pixels and the solver is run for each individual block. Therefore each individual block has its own solution and the program doesn't require synchronization. The padding strategy improves the quality and performance significantly. Using the local solver our reconstruction became 24 times faster than the global solver. The details can be found at chapter 5.

1.4 Organization

The remainder of this thesis is organized as follows. In chapter 2, a review of relevant prior art is done. Specifically, a review of the background behind volume rendering, exploring the current volume rendering tools and also presenting an overview of GPU programming. Later in Chapters 3 and 4, the implementation techniques are described. In Chapter 3 we highlight the global solver while Chapter 4 focuses the detailed mechanism of the local solver. In Chapter 4 an examination of an adaptive sampling strategy is also performed. The experimental results are in Chapter 5 and Chapter 6 discusses the limitations of this study and its future directions.

Chapter 2

Related Work

This chapter is focused on ray casting based volume rendering techniques that take advantage of parallel processing on the GPU. It will begin with a brief discussion of volume rendering and highlight the growing demand of GPU based volume rendering; then techniques for compressive volume rendering will be reviewed. A discussion of the differences between CPUs and GPUs is then provided to show the advantages that the GPU offers – massive parallelism by providing parallel programming models. The chapter will conclude with an overview of using CUDA (Compute Unified Device Architecture) for parallel GPU based programming.

2.1 Volume Rendering

Volume rendering deals with the visualization of 3D scalar fields; and these 3D scalar values are assigned to every point in a continuous 3D space. These 3D scalar values can be produced by simulation or by some data acquisition techniques. In the medical field, these values are obtained by imaging techniques such as computed tomography (CT) or magnetic resonance imaging (MRI). The process of projecting a 3D scalar field to 2D image space is known as volume rendering. Visualizing volumetric data plays an important role in scientific visualization in the fields of medicine, biology and engineering. The most common volume rendering model used in computer graphics and the scientific visualization community is proposed by Levoy et al. [7]. This model renders volume data into the display screen by sampling the volume at regular intervals, assigning optical properties to data values and composing the results using over or under operators [8].

2.1.1 Volume Data

The data acquired through CT, MRI or simulation can be thought of as a three-dimensional array of sample values. Each value is stored at the corner of a voxel of a cuboid; where the length, width and height of the voxels depend on the sampling distance in the x,y and z dimensions respectively. In most cases, these samples are taken at regularly spaced intervals along the three orthogonal axes. There are also other types of grids used such as rectilinear grids, curvi-linear grids, and unstructured grids. The vast majority of volumetric data are however acquired on Cartesian grids and for this reason, the focus of this thesis will be on the Cartesian grid.

2.1.2 Ray Casting

One of the most commonly used direct volume rendering techniques is ray casting. According to Levoy [7], volume data is rendered into the image plane by assigning optical properties and composing the results using over and under operators [8]. In this approach, there is a source of rays, a virtual camera and a volume data set. The rays are cast towards the volume from the origin where the volume is considered to be loaded in world space. For every pixel, rays are cast and these rays traverse through the volume. Whenever a ray intersects the volume at a point, color and opacity are assigned using a transfer function. As the ray continues to intersect the volume, it keeps accumulating colors; at the end the accumulated color is displayed on the screen. A traditional ray casting technique is depicted in figure 2.1 and the steps mentioned by Levoy et al [7] are displayed in figure 2.2.

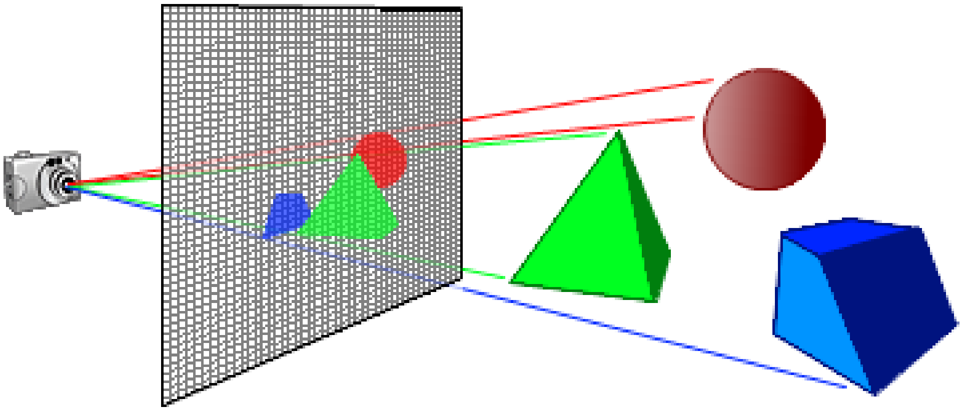


Figure 2.1: Ray casting through a volume [2]

The modern direct volume rendering pipeline that was proposed by Levoy [7] has five stages:

1. Pre-processing: This stage converts the data into a form suitable for rendering images. This includes data filtering, resizing, cropping, bricking and derivative estimation.

2. Reconstruction: As the ray steps through the volume, an actual sampled value or interpolated value from neighbouring voxels may be obtained. The choice of step size has a profound effect on the image quality and speed of the volume renderer. Smaller step sizes produce better quality images but take longer to render; larger step sizes take less time to render but a diminished image quality is obtained. Another factor affecting the image quality and rendering times is the method of interpolation chosen. Tri-cubic interpolation is used to achieve high quality images; this involves 64 neighbouring voxels which takes significantly more time than tri-linear interpolation which uses 8 neighbouring voxels.

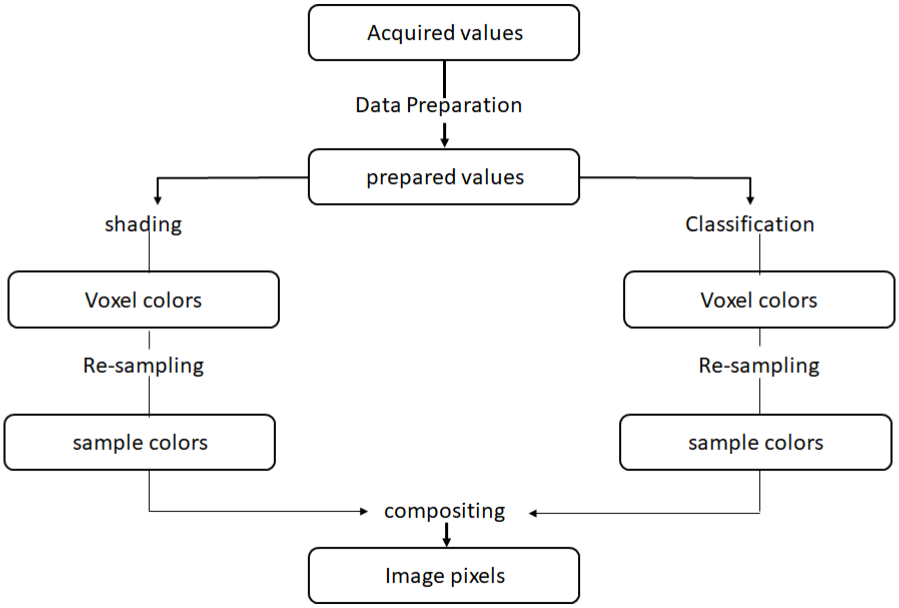


Figure 2.2: Stages of volume rendering

3. Shading: The acquired sample values are mapped via a transfer function to produce color; this color keeps accumulating until it reaches a predefined threshold and at the end the final color is assigned to an individual pixel. Collectively these pixel values form the shape of the volumetric object. There are various shading models in use but the most commonly used shading technique is proposed by Phong [9].

4. Classification: This stage is used to classify different regions of interest of the volume object. The voxels in a volume contain certain properties that reside as numerical values; visual properties are assigned to the numerical values to represent specific properties. An example is generating an iso-surface for a particular constant isovalue from the volume.

5. Compositing: All the assigned colors are accumulated along the path of ray traversal and projected to the 2D image plane with the output being the final color of the pixel.

2.1.3 Volume Visualization Tools

With the increasing demand in 3D visualization, there are many volume visualization tools available such as VTK [10], ParaView [11], Visit [12] and Voreen [13]. These tools were created for the scientific visualization community and are open source. Visualization Toolkit (VTK) [14] is a free software for image processing and visualization and is one of the oldest applications used for visualization and image processing. VTK supports a wide variety of visualization algorithms and it also supports parallel processing.

ParaView [15] is a multi-platform visualization application that uses VTK as its rendering and data processing engine. ParaView employs a unique parallel rendering algorithm to speed up the volume rendering process [16] and it also has a user friendly graphical user interface to easily edit the transfer function.

Though the above mentioned tools are used widely in scientific community, they don't allow the user to customize all the options. As for example for high quality rendering these tools don't provide the option to choose between different interpolation methods. Moreover the step size through the volume can't be changed. On the contrary, in my work user can customize all these options.

2.2 Literature Review

2.2.1 GPU based volume rendering

Real-time interactive volume rendering is one of the major research goals in the scientific visualization community. The key features of a real-time volume rendering system [17] are listed below:

- Visualizing rapidly changing data-sets.
- Real-time exploration of 3D data-sets.
- Interactive manipulation of visualization parameters.

GPU accelerated volume rendering has become one of the standard approaches for real-time interaction. In recent years, most volume rendering techniques are based on ray casting. GPU based ray casting stores the volume in texture memory to exploit the GPU's built in interpolation techniques. The first published texture based ray casting method was proposed by Qu et al. [18]. Basic ray casting by leveraging texture memory was first introduced by Cullip and Neumann et al. [19] and Cabrel et al. [20]. Many extensions of this simple texture based volume rendering have been proposed that take advantage of more advanced texture mapping capabilities to increase interactivity [21]. Donelly [22] proposed that by using empty space skipping, by virtue of distance functions encoded in 3D-textures, it would accelerate ray traversal. The dramatic increase in using texture memory requirements made the technique infeasible for large scale data. Later methods proposed by Dummer [23] and Policapro et al. [24] significantly reduced the texture memory requirement. Kruger and Westermann et al. [25] proposed another acceleration technique based on early ray termination and empty space skipping into texture based volume rendering. This proposal contains a stream model for volumetric ray casting that exploits the intrinsic parallelism and efficient communication on modern graphics chips. Cohen and Shaked [26] proposed an accelerated ray casting approach based on a hierarchical pyramidal data structure. This accelerated the ray traversal by employing a maximum-quadtree; storing the maximum height displacement value of areas covered by its nodes in order to identify larger portions of the height field not intersected by the ray.

Interactive volume rendering spurred a growing interest in classifying the region of interest for the volume data set; however the user must define how the different parts of the volume will be visualized. One way of visualizing the region of interest is using transfer functions [27]. Levoy et al. [7] introduced 2D transfer functions but the use of multi-dimensional transfer function proposed by Kniss et al. [28] greatly improved rendering quality. The pre-integrated volume rendering method proposed by Engel et al.[29] also improves rendering quality. A slice based volume rendering approach was proposed by Swan and Yagel et al. [30] to overcome the trade-off between image quality and the speed of volume rendering. A ray casting algorithm for structured and unstructured grids that does not rely on volume slicing has been presented in [31]. An implementation of basic volume rendering with functionality that does not use any data structure or hierarchical data or multiple rendering passes has been shown by NVIDIA in [32].

2.2.2 Compression domain volume rendering

Hardware supported texture mapping has become one of the most widely used operations to achieve interactivity in advanced rendering techniques. Graphics chip manufacturers have devoted themselves to design real-time 2D and 3D texture mapping that can significantly increase performance; however the limited size in texture memory becomes one of the major hurdles to the application developers. One way to address this problem was proposed by Guthe et al.[33] by using a wavelet transform to significantly compress input data to a hierarchical wavelet representation. Although the method was able to interact with large data sets it still required decoding to every texture tile before they are transferred to the GPU. Hierarchical data structures have been proposed by Westermann and Ertl [34]. Shen et al.[35] substantially improved the volume rendering speed by creating a new data structure called time-space (TSP) tree. Unfortunately none of the techniques were able to decode the data on the GPUs. The first time volumetric data was decoded and rendered in a GPU was proposed by Lum et al. [36] and this approach was well suited for time varying data. Engel et al.[37] presented an implementation for static data sets that provides high quality images for low resolution volumetric data. Schneider et al. [38] outlined hardware accelerated texture decoding, rendering and also provided a basis for volumetric texture compression. Another multi-resolution compression-domain GPU volume rendering design was proposed by Gobbetti et al. [39]. In this approach, compressed models were streamed on the fly and loaded on demand; and to describe the empty space regions above the height field they calculated cone ratios for each cell allowing for fast search convergence during ray traversal.

Compressive rendering is a term coined by Sen and Darabi [40]. Compressive rendering allows the user to reconstruct the full image using only the subset of the image pixels to make the rendering process more efficient. Liu and Alim [4] investigated four different image order techniques for compressive volume rendering. The smoothness-based techniques significantly outperforms the other methods that are based on compressed sensing and are also robust in the presence of highly incomplete information.

In this thesis the smoothness based- technique is applied as it generates the most optimal results. This same technique is used in conjunction with a parallel strategy so that it can be used in real time applications.

2.3 CPUs vs GPUs

Today's applications are developed using both CPUs and GPUs to gain high efficiency; as a result heterogeneous computing is gaining popularity day by day. The throughput oriented architecture of GPUs distinguishes it from the architecture of CPUs. Figure 2.3 shows the differences in the architecture of CPUs and GPUs in terms of cache or local memory, registers, Single Instruction Multiple Data (SIMD) units, and the control logic unit. GPUs have a larger number of registers to support the large amount of threads whereas CPUs can only handle a small number of threads. GPUs also tend to have a larger number of SIMD units compared to CPUs. The CPU control logic unit is sophisticated compared to a GPU's control logic unit which is simple to manage. In general, the CPU's latency orientated architecture was designed with the primary goal of executing as many instructions as possible belonging to a single serial thread within a given time frame [41]. In contrast, GPUs are designed with the assumption that they will be presented with workloads in which parallelism is prevalent [42].

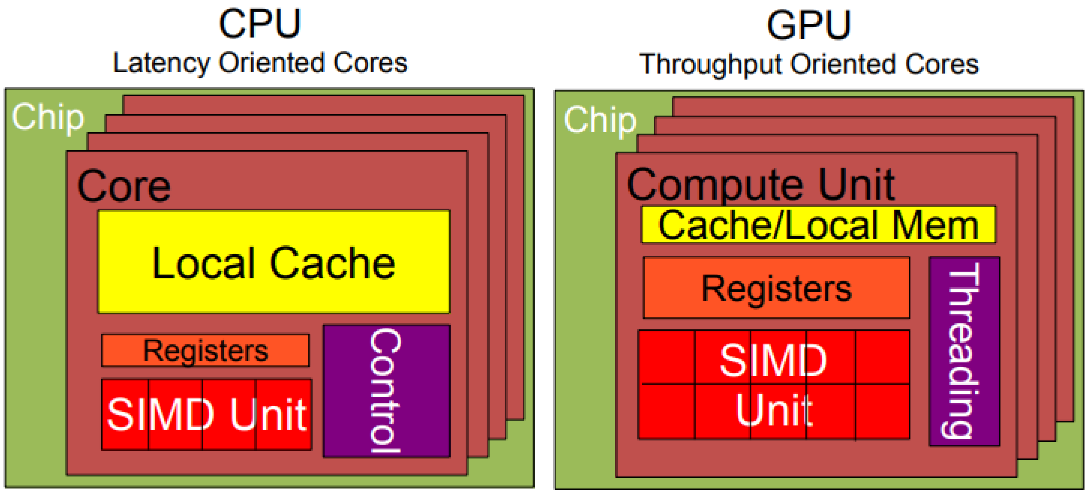


Figure 2.3: Higher level comparison of CPU and GPU [1]

There three prominent features in CPU and GPU design are the following.

- **ALU:** Arithmetic Logic Units are designed according to a powerful logic that can generate numerical arithmetic results in very few clock cycles. The CPUs have extremely short latency producing a 64 bit floating point of the arithmetic values. GPUs come with a larger number of latency ALUs; these ALUs are heavily pipelined to accept operations in every clock cycle and will take a longer time to produce results. Therefore having multiple ALUs will allow a large number of threads can perform arithmetic operations simultaneously.

- **Caches:** CPUs tend to have larger caches which are designed to convert long latency memory access to short latency cache accesses. The strategy is to keep as many data elements as possible in the caches; so that whatever CPU execution units need to access data, they will be found in the cache from the previous access. GPUs are comprised of very small sized caches that do not retain data from its previous access. These caches are designed for staging units for a large number of threads. The numerous threads accessing the same data simultaneously is handled by cache memory consolidating all the requests into one that is directed to the DRAM. When the data comes back, the cache controller works by forwarding logic to distribute the data to all threads.

- **CU:** The CPU's sophisticated Control Unit comes in two forms. One has branch prediction for reduced branch latency and the other one is data forwarding for reduced data latency.

  The GPU's CU is a simple control unit as shown in Figure 2.3. In GPUs we do not have any branch prediction.

2.4 Parallel Programming Model

A Parallel Programming Model is an abstraction of parallel computing architecture [43]. Parallel programming models involve a heterogeneous parallel programming interface that enables the exploitation of data parallelism by using both CPU and GPU. Examples of parallel computing models include:

1. Message Passing Interface (MPI)
2. Open Multi-Processing (openMP)
3. General Purpose Graphics Processing Unit (GPGPU) with CUDA or OpenCL

The idea behind data parallelism is that different parts of the data can be processed simultaneously and independently of each other; a very simple example of data parallelism is vector addition. In vector addition, two different vectors are added together and in the resultant vector the individual elements are added together. If there are a large number of elements in the vector and there is a large amount of hardware, all the operations should be be performed in parallel fashion. This is how we achieve high performance using the parallel programming model.

2.5 CUDA Programming Model

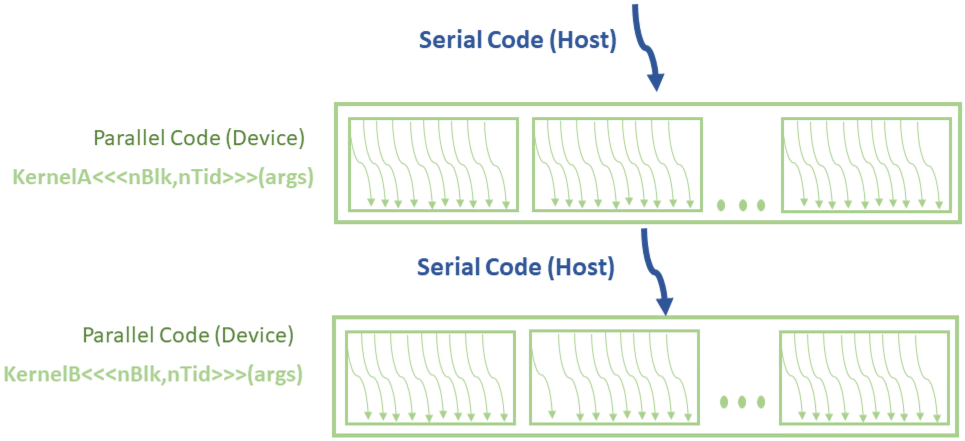


Figure 2.4: CUDA Execution Model

CUDA is a parallel computing platform and programming model introduced by NVIDIA [44]. The parallel execution model of CUDA is based on the host and the device. The host refers to the CPU and the device refers to the GPU. When an application is executed the serial parts are executed through the host and the parallel parts are executed through the device. The parallel execution part is written using a specialized function called *kernel* functions. In Figure 2.4, there are two kernel functions shown called *kernelA* and *kernelB*. CUDA kernal functions also take arguments like the C programming language. CUDA kernals also take additional parameters called configuration parameters; these parameters are wrapped by three less than signs at the beginning and three greater than signs at the end. In the CUDA programming model, the number of threads per block and the number of blocks per grid have to be mentioned. The thread and block configurations are put inside the less than and greater than signs separated by a comma. As the code is executed; the serial portions are executed first by the host side. Then *kernelA* will be executed by the device and the control will be returned to the host side. This is how the execution order will keep changing between the host and the device.

2.5.1 CUDA Threads

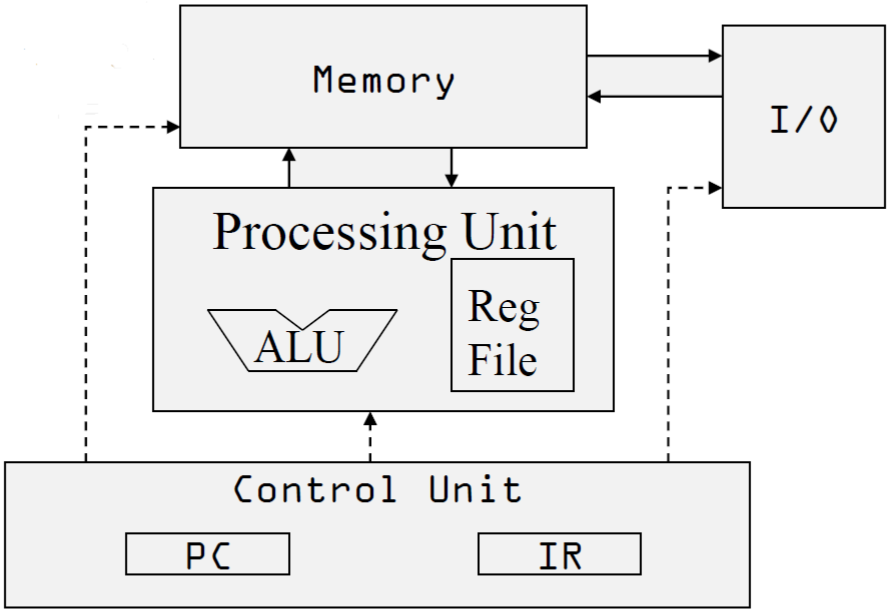
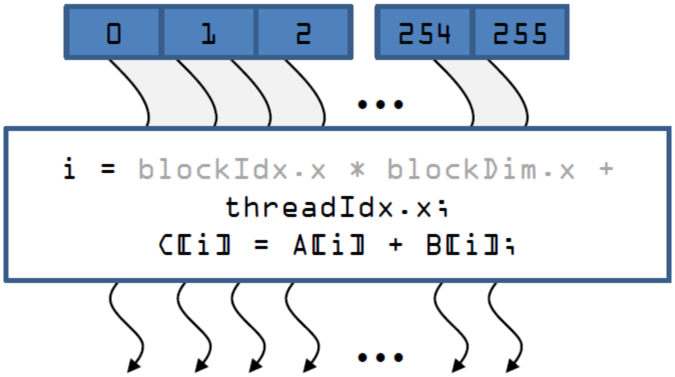


Figure 2.5: A Von-Neumann Processor

Figure 2.5 shows a simplified diagram of a Von-Neumann processor model proposed in the 1940s. All the modern processors designed today are a variation on this model. The program counter in the CU finds the instruction in memory and the memory returns the instruction as a bit which is stored in the Instruction Register. These actions are controlled by a signal which determines the activities in the ALU and Register files. These activities may also involve the memory of the I/O access operation. Finally some of the data will be moved between the memory and I/O. A CUDA thread is also a virtual Von-Neumann processor which are able to execute programs. CUDA kernal functions and hardware will generate a large number of Von-Neumann processors that will be executing the same kernal function hence the name *Single Instruction Multiple Data* (SIMD).



```
i = blockIdx.x * blockDim.x +
        threadIdx.x;
    C[i] = A[i] + B[i];
```

Figure 2.6: Arrays of parallel threads

The execution of CUDA kernels actually launches a grid of array of threads; which can be 1D, 2D or 3D and all these threads have their own unique index. This index is represented as *thread1dx.x* in CUDA where x represents the x-direction. In Figure 2.6, it shown that there are 256 threads and their indices are from 0 to 255. Each thread index is used to calculate the memory address shown in Figure 2.6 by a box. In this example, a calculation of variable i will generate an address which is private to every thread. During the execution time of the statement $C[i] = A[i]+B[i]$, the i value for every thread will be unique.

2.5.2 CUDA Blocks

The previous section shows how an individual CUDA thread block functions; which can be used to extend a similar idea to multiple blocks of threads. Figure 2.7 displays an array of threads organized into blocks and all the threads have a thread index and a block index. The block index variable is represented as *blockIdx.x*. The thread index and block index are used to form an address that corresponds to a memory location. In this example block 0 has a thread with index 0, similarly block 1 has a thread with index 0; but the i value is not going to be the same. For the first block (block index 0) *blockidx.x* is 0 but for block 1, *blockidx.x* is 1. This is how threads with the same thread id in different blocks form a uniform coverage for all the elements in the array.

Threads within a block can communicate through shared memory and they can also cooperate in atomic reduction operation and barrier synchronization.

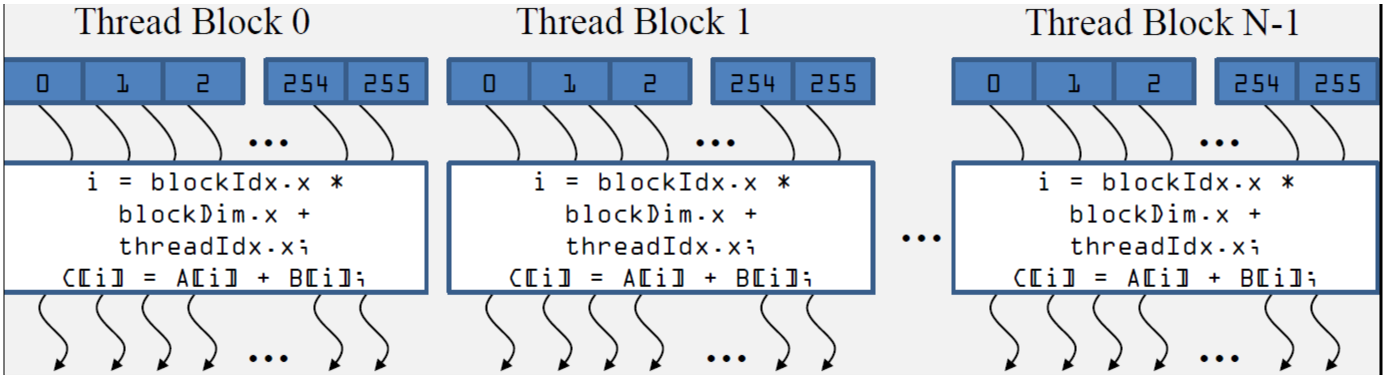


Figure 2.7: Thread Blocks

2.5.3 CUDA Memory

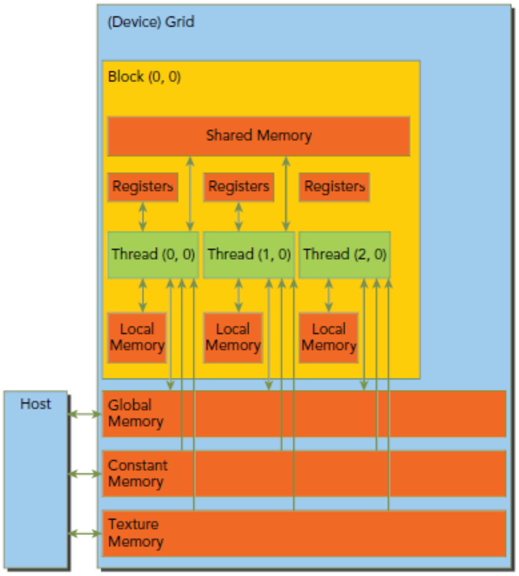


Figure 2.8: CUDA memory model [3]

To achieve high memory efficiency, CUDA has different memories for programmers to use. Host memory refers to the memory attached to the CPU and device memory is attached with the GPU which can only be accessed by threads. It must be noted that data must be copied to the device memory at the beginning of every CUDA application.

Memory	**Location**	**Cached**	**Access**	**Scope**	**Lifetime**
Register	cache	n/a	Host: none, Kernel: R/W	thread	thread
Local	device	Yes	Host: none, Kernel: R/W	thread	thread
Shared	cache	n/a	Host: none, Kernel: R/W	block	block
Global	device	Yes	Host: R/W, Kernel: R/W	application	application
Constant	device	Yes	Host: R/W, Kernel: R	applicaiton	applicaiton

Table 2.1: Overview of different types of GPU memory

The GPU provides various types of memory that provide some built in mathematical functions but also faster access compared to CPU memory. A high level overview of CUDA memory is shown in Figure 2.8. The types of memory available to a CUDA programmer are:

1. Register memory: All the scalar variables are stored in fast registers. The registers are fast but they are private to the thread and users can read/write on the registers.

2. Local memory: This memory is used only when primitives are not inside the registers; this is part of global memory.

3. Shared memory: The scope of shared memory is limited within a block and the maximum size of shared memory per block is usually very small (currently 48KB per streaming processor). Shared memory is very fast and is also subject to bank conflicts. To achieve high memory bandwidth, shared memory is divided into equal sized memory modules called banks that can be accessed simultaneously. If multiple memory requests map to the same memory bank, then accesses become serialized and performance degrades; this is called bank conflict. Shared memory can be read/written by threads assigned for a particular block but it must be synchronized after each time writing.

4. Constant memory: Constant memory is very fast but it is a write once type memory.

Constant memory can be directly written from the CPU using the API *cudaMemcpyToSymbol()*. The scope of constant memory is the lifetime of the program.

5. Texture memory: Another type of read-only memory that is available for using in programs written with CUDA C is Texture memory. Although NVIDIA designed the texture units for the classical OpenGL and DirectX rendering pipelines, texture memory has some important properties that make it extremely useful for computing. Texture memory is cached on chip, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM [45].

   Texture memory is ideally suited for storing and accessing volumetric data. There are some other features of texture memory for programmer convenience such as:

   (a) interpolation (nearest neighbor, bilinear or trilinear) of adjacent data values,

   (b) automatic normalization of data when fetched:

   - for unsigned values: [0,1],
   - for signed values: [-1,1];

   (c) automatic normalization of array indices, and

   (d) automatic boundary handling.

The overview of different GPU memories is displayed in Table 2.1. The performance of an application that uses GPUs is greatly influenced by the type of GPU memory used. A discussion of the comparison of using different types of GPU memory appears in the following chapters.

Chapter 3

Compressive Volume Rendering

3.1 Overview

The concept of compressive rendering allows us to render a full image by using only a subset of the pixels in the image. The idea is very simple and straightforward. As rays are traced rays for a small subset of the pixels, then the full image is reconstructed using an appropriate image representation model. For compressive volume rendering, the idea initially proposed by Sen and Darabi [40] is extended where they presented a two-step algorithm for compressive rendering:

- Step One: Render k pixels out of n. The rendered pixels are determined using a low-discrepancy distribution such as the Poisson-disk distribution.

- Step Two: The final image is reconstructed using the compressed sensing framework.

The proposal by Sen and Darabi [40] theorizes that casting rays for some of the pixels and later using a numerical technique to recover the missing pixels to finally render an image. This work uses compressed sensing but later Liu and Alim et al. [4] have shown that the smoothness based approach performs better.

The thesis methodology uses these two steps in the proposal by Sen and Darabi [40] with required modifications. These modifications includes the use of the linear pixel shuffling algorithm proposed by Anderson et al. [46]. instead of the Poisson disk pattern; finally the solution based on smoothness proposal by Liu and Alim [4] is used to recover the missing pixels.

3.2 Problem Definition

Let $\mathbf{x}$ denote the rendered image, which has N pixels, where $N = W \times H$. W and H represent the width and height of the image respectively. For simplicity, we consider $\mathbf{x}$ to be a linearized representation of the image, i.e. $\mathbf{x} = [x_1, \ldots, x_N]^T$. Instead of rendering all the pixels in $\mathbf{x}$, the focus is in rendering a subset of the pixels. The rendered pixels are given by:

$$\mathbf{y} = \mathbf{S}\mathbf{x} \tag{3.1}$$

where, $\mathbf{y} = [y_1, \ldots, y_M]^T$ is an column vector with the dimension $M \times 1$. It is to be noted that $M << N$. $\mathbf{S}$ is a $M \times N$ binary sampling matrix which contains information about the contributing pixels. Each row of this matrix contains only one 1 (corresponding to the pixel location that is to be rendered) and the rest of the row is filled with 0's.

The problem can be seen as a scattered data fitting problem [47]. The goal is to find a smooth solution in a prescribed image space. Let $j_1, \ldots, j_N$ denote the 2D pixel locations in $\mathbf{x}$ and let $k_1, ..., k_N$ denote the 2D pixel locations in $\mathbf{y}$. The goal is to find the coefficients $c = [c_1.....c_N]^T$ for the approximation:

$$f(t) = \sum_{n=1}^{N} c_n \varphi(t - j_n), \tag{3.2}$$

such that $f(k_m) \approx y_m$ for $m = 1, \ldots, M$. Moreover, it is desired that the function $f(t)$ be smooth. The second-order Beppo-Levi seminorm provides a useful notion of smoothness. For functions $g(t)$ and $h(t)$, the second-order Beppo-Levi inner-product is defined as:

$$\langle g, h \rangle_{BL_2} := \langle \partial_{t_1 t_1} g, \partial_{t_1 t_1} h \rangle + 2\langle \partial_{t_1 t_2} g, \partial_{t_1 t_2} h \rangle + \langle \partial_{t_2 t_2} g, \partial_{t_2 t_2} h \rangle \tag{3.3}$$

where $\langle \cdot, \cdot \rangle$ represents the standard L_2 inner-product. For real-valued continuous functions a and b, it is defined as

$$\langle f, g \rangle = \int_{\mathbb{R}^2} f(x) g(x) \, dx \tag{3.4}$$

The BL_2 seminorm is defined in the continuous domain and measures smoothness via the second-order derivatives. Smooth functions have a low BL_2 norm and vice-versa. The minimization problem can be formulated as:

$$\min_f \lambda \, \|f\|_{BL_2}^2 + \sum_{m=1}^{M} (f(k_m) - y_m)^2 \tag{3.5}$$

where the minimization is to be carried out over all functions f that are of the form shown in Eq. 3.2. The first term in the above equation measures the smoothness of the solution $f(t)$ by the energy present in all of its second derivatives, and the second term is a fidelity term that attempts to fit the function $f(t)$ to the available data.

To solve this minimization problem, we need to choose a kernel function $\varphi(t)$. The size of the kernel function affects the performance of the solver. With bigger size kernels, computation becomes expensive. To ensure consistency with the recovery methods and to obtain good quality approximations, we choose the optimal interpolating cubic B-spline proposed by Blu et al. [48] is chosen. It is defined as

$$\beta_I^3 := \beta^3(t) - \frac{1}{6}\frac{\partial^2}{\partial t^2}\beta^3(t) \tag{3.6}$$

where $\beta^3(t)$ is uni-variate uniform centered cubic B-spline. Tensor product, i.e $\varphi(t_1,t_2) = \beta_I^3(t_1)\beta_I^3(t_2)$ provides the corresponding bi-variate kernel. By following [47], our problem can be written as:

$$\min_{\mathbf{x}} \|\mathbf{S}\mathbf{x} - \mathbf{y}\|_2^2 + \lambda \mathbf{x}^T \mathbf{H}\mathbf{x} \tag{3.7}$$

where $\mathbf{H}$ is a convolution matrix and is defined as:

$$H_{p,q} = \langle \varphi(\cdot - j_p), \varphi(\cdot - j_q) \rangle_{BL_2}. \tag{3.8}$$

The matrix-vector product $\mathbf{Hx}$ can equivalently be obtained by convolving the image with a 2D convolution filter F whose entries (following Eq. 3.8) are given by:

$$F = \begin{vmatrix} 11/151200 & 47/4725 & -38/479 & -53/1890 & -38/479 & 47/4725 & 11/151200 \\ 47/4725 & -533/3150 & 206/315 & 191/189 & 206/315 & -533/3150 & 47/4725 \\ -38/479 & 206/315 & 223/1229 & -3950/493 & 223/1229 & 206/315 & -38/479 \\ -53/1890 & 191/189 & -3950/493 & 3158/135 & -3950/493 & 191/189 & -53/1890 \\ -38/479 & 206/315 & 223/1229 & -3950/493 & 223/1229 & 206/315 & -38/479 \\ 47/4725 & -533/3150 & 206/315 & 191/189 & 206/315 & -533/3150 & 47/4725 \\ 11/151200 & 47/4725 & -38/479 & -53/1890 & -38/479 & 47/4725 & 11/151200 \end{vmatrix}.$$

As Eq. 3.7 involves the l_2 norm, the following least-squares problem is obtained by differentiating with respect to $\mathbf{x}$:

$$(\mathbf{S}^T\mathbf{S} + \lambda\mathbf{H})\mathbf{x} = \mathbf{S}^T\mathbf{y}. \qquad (3.9)$$

Since, the matrix $(\mathbf{S}^T\mathbf{S} + \lambda H)$ is symmetric and positive definite, equation 3.9 can be efficiently solved by using the *Conjugate Gradient Solver* (CGS). Let us denote $(\mathbf{S}^T\mathbf{S} + \lambda H)$ as $\mathbf{A}$ and $\mathbf{S}^T\mathbf{y}$ as $\mathbf{b}$. The procedure of the CGS (obtained from [49]) is shown in Algorithm 1.

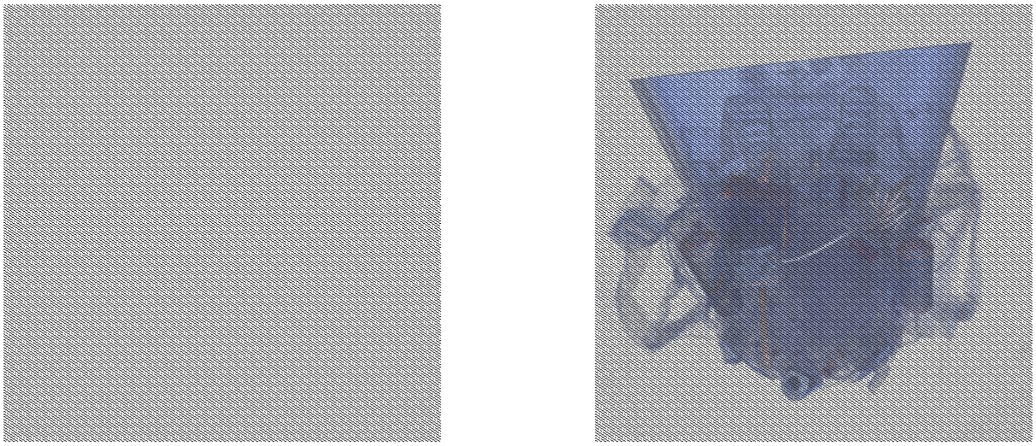
Algorithm 1 Conjugate Gradient Solver

```
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
repeat
    $\alpha_k := \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
    $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$
    if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop
    $\beta_k := \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$
    $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
    $k := k + 1$
end repeat
The result is $\mathbf{x}_{k+1}$
```

3.3 Recovery of Missing Pixels using CUDA

For the reconstruction of missing pixels the conjugate gradient solver is used. The solver is computationally expensive and the performance of the solver also depends on the size of the image and the number of iterations. However experimental results shown in [4] were not suitable for real-time rendering due to the performance. The main focus in this thesis is to optimize the performance while maintaining the same quality. To gain performance efficiency, the CUDA's parallel programming model is exploited. In this problem, image data, where many arithmetic and logical operations need to be performed concurrently are dealt with. CUDA uses *Single Instruction Multiple Thread* (SIMT) model. SIMT is an extension of *Single Instruction Multiple Data* (SIMD). The reconstruction kernel launches a number of threads that enter the same program together and these threads are dynamically scheduled onto a SIMD data path such that the threads are executing the same instructions that are scheduled concurrently.

3.3.1 Initialization



a. Mask using 70% pixel b. Rendered volume using 70% pixel

Figure 3.1: Binary mask and Rendered image with missing pixel using the mask

Using the pixel shuffle pattern proposed by Anderson et al. [46], a binary mask is generated. This mask gives the locations for casting rays. We consider this binary mask as a 1D vector and use it to rendered the volume. This is the input for our conjugate gradient solver. We have to recover the unknown pixels using our solver. Both the mask and volume rendered image using the mask are shown in Figure 3.1.

3.3.2 Reconstruction via a Global Solver

The contribution of this chapter is a parallel implementation of the conjugate gradient solver. In the global solver, our implementation involves CUDA's built in library to perform the FFT (fast Fourier transform) based convolution and vector dot product operations. Our methodology involves the following steps:

Step 1: Calculating Ax

One of the major tasks of our solver is to calculate $\mathbf{Ax}$. We have $\mathbf{A} = \mathbf{S^T S} + \lambda \mathbf{H}$; hence, $\mathbf{Ax}$ becomes $\mathbf{S^T Sx} + \lambda \mathbf{H}\ \mathbf{x}$. Calculating $\mathbf{Ax}$ involves the following operations:

- Convolution Operation.
- Dot product operation.
- Vector multiplication.
- Vector Addition.

After calculating $\mathbf{Ax}$, the calculation of $\mathbf{r}_0$ is obtained by subtracting $\mathbf{A}$ from $\mathbf{b}$. Note that $\mathbf{b} = \mathbf{S^T Y}$. We initialize $\mathbf{p}_0$ with $\mathbf{r}_0$. We store the result of $\mathbf{A}$ to avoid repeating calculations.

Step 2: Calculating k^{th} step value of α

α value calculation requires division operation. The numerator is the dot product of $\mathbf{r}_k^T$ and $\mathbf{r}_k$. The denominator is another dot product between k^{th} value of p and stored value of A.

Step 3: Calculating $(k+1)^{th}$ step value of x

This step requires multiplication with α and vector $\mathbf{p}_k$. Then vector addition operation between k^{th} value of $\mathbf{x}$ are performed and the resultant multiplication is obtained.

Step 4: Calculating $(k+1)^{th}$ step value of r

This is similar to step 3 except instead of addition it needs subtraction operation.

Step 5: Calculating k^{th} step value of β

Calculation of β requires two dot products and one division. In this case, the numerator is the vector dot product of $\mathbf{r}_{k+1}^T$ and $\mathbf{r}_{k+1}$. The denominator is the dot product of $\mathbf{r}_k^T$ and $\mathbf{r}_k$.

Step 6: Calculation of $\mathbf{p}_{k+1}$: This is similar to the operations mentioned in step 3.

These are the steps for finding the solution of our problem. The final output is $\mathbf{x}_{k+1}$. Our solver finds the residual on each iteration. We can fix the number of iteration or we can put a conditional flag to stop the repetition. The number of iteration should be chosen carefully since it has a significant impact on the performance of the solver. Many iteration may take longer time while producing better result. On the other hand, a fixed small number of iterations can deteriorate the quality of the reconstruction.

3.3.3 Convolution

Among all the operations, Convolution is the most expensive one. To reduce the computational cost, we used CUDA's FFT library for the convolution which is developed by V Podlozhnyuk [50]. The general steps for the convolution are:

- Create and initialize the plan for both forward and inverse FFT.
- Usually the filter size is smaller than the image size. It is necessary to pad the filter with the same image size.
- Apply forward FFT to the input data and the filter.
- Perform the point wise multiplication for the FFT of input data and the FFT of padded filter data.
- Apply inverse FFT on the result of the multiplication

Using these steps, the outcome of $\lambda\mathbf{Hx}$ is obtained. The result is written into global memory and once the result writing operation is done the FFT plan that is created must be eliminated. Our next step is to calculate $\mathbf{S}^T\mathbf{Sx}$

3.3.4 Vector multiplication

CUDA manages parallelism by executing warps of threads where each warp consists of 32 threads where threads are launched corresponding to the image size. Each thread index is the address of that particular data in an $1D$ array. Every thread is running simultaneously while performing a multiplication function. Once the multiplication is done, each thread writes the result to its corresponding location. The size of binary sampling matrix $\mathbf{S}$ is $(imagewidth \times imageheight)$ by

$(imagewidth \times imageheight)$. The **Sx** operation is a down sampling operation where pixels are dropped. Furthermore the $\mathbf{S}^T\mathbf{Sx}$ operation inserts 0 in the location of missing pixels; this massive calculation is done only in few milliseconds using CUDA.

3.3.5 Vector Addition/Subtraction

Once calculating $\lambda\mathbf{Hx}$ and $\mathbf{S}^T\mathbf{Sx}$ are finished, the next step is to perform vector subtraction. By using CUDA's thread configuration, every thread subtracts $\lambda\mathbf{Hx}$ from $\mathbf{S}^T\mathbf{Sx}$ and stores the result in global memory.

3.3.6 Vector Dot Product

NVIDIA's CUDA also provides a built-in optimized function to calculate vector dot products. CUDA's **CUBLAS** library to calculate dot-product operations between vectors. In order to calculate the dot product, a dot product handler is created first. The dot product of any arbitrary sized vectors is calculated using the **cublasSDot** function with necessary parameters and this result is written into the device memory.

Even though the quality of the reconstructed image is acceptable, the global reconstruction process is too slow to interact with the volume renderer (quality and timing results are provided in Chapter 5). This time doesn't include the volume rendering time. Though the FFT based convolution is fast, it requires memory to be copied back and forth between the CPU and GPU and therefore decreases performance. Furthermore, the *dot product* operation using **cuBlas** library also requires memory copy for each iteration, which also significantly reduces the performance of our solver.

Chapter 4

CGS based Compressive Volume Rendering via a Local Solver

The outcome of reconstructing the volume rendered image globally; with missing pixels, using the parallel strategy is met with mixed results. The image quality is acceptable but the performance is poor due to the high number of data transfers between the CPU and the GPU. In order to have an interactive volume renderer, the parallel strategy is not suitable as it is taking significantly longer to reconstruct the image. To optimize the solver, a method is required where there is less memory traffic to deal with. Global memory access is always a bottleneck for performance; one way to reduce the memory access is to use tiling. In this instance, tiling means partitioning the data into smaller blocks. At the block level the entire computation does not require any global memory access; instead CUDA's shared memory is used. The result is stored in shared memory which is much faster than the use of the regular global device memory. The performance is also optimized by using block-wise convolution and vector dot product reduction operation.

4.1 Optimizing CUDA application

Reviewing Equation 3.9 from Chapter 3, we have

$$(\mathbf{S}^T\mathbf{S}+\lambda\mathbf{H})\mathbf{x}=\mathbf{S}^T\mathbf{y}.$$

To solve this equation using a conjugate gradient solver on the block level; the algorithm for the gradient solver is also described in Algorithm 1. To optimize the performance of the solver the following points must be addressed:

Thread Configuration: In any CUDA application, the thread configuration is very important because performance of any CUDA application depends on it. The more threads we can occupy, the better better the performance. Threads are organized in warps which usually consist of 32 threads. The block sizes that are not multiples of 32 are padded with threads to fill up the warps.

Global Memory Bandwidth: CUDA kernel performance also largely depends on accessing data from the global memory which is always costly. The tiling process can reduce the cost of computations; which is discussed later in this chapter. Another optimizing strategy is using memory coalescing.

Dynamic Partition of Resources: One of the optimizing strategies is to have the multiprocessors fully occupied. Threads are assigned to thread blocks during execution. Each CUDA device has its limitation on the number of threads per *streaming multiprocessor* (SM) and each SM can accommodate a fixed number of blocks. Therefore the resources must be partitioned to achieve optimal utilization.

Data Pre-fetching: Global memory comes with limited bandwidth while accessing data. The tiling strategy can address this issue but cases may arise where threads with very small independent instructions are waiting for their access result which can be handled using shared memory. In this instance data is pre-fetched and stored in shared memory and is also faster than global memory.

4.2 Local Conjugate Gradient Solver

In Equation 3.9, there are 4 components. The required result $\mathbf{x}$, the sampling matrix $\mathbf{S}$, convolution filter $\mathbf{H}$, and the given image with missing pixels $\mathbf{y}$. In Chapter 3, the implementation bottleneck of our global solver is discussed. The global problem can be decomposed into local problems. Since there will be computations performed on the image block it is necessary to partition it into smaller blocks; and each of these blocks will be mutually exclusive. Instead of running the convolution for the entire image we can run convolution on the individual image blocks which will reduce data traffic significantly and also the usage of shared memory will significantly improve the performance. After partitioning the data, instead of running the solver for the entire image, it will be run locally for each block thereby giving each block its' own solution. It is to be noted that, we are running the same conjugate gradient solver at the block level. Since the convolution matrix is very local and the partitioned padded region has the neighbour information, the local solver doesn't suffer from incoherence issues.

4.2.1 Background

In the global solver the vector size is $imageWidth \times imageHeight$. After converting into local problem the vector size is $blockWidth \times blockHeight$; where the $blockWidth$ and $blockHeight$ are the partitioned image's block width and height respectively. At the block level, the solver runs only one block resulting in better performance. The use of shared memory significantly improves the performance of the solver as well. At the block level, the same algorithm (Algorithm 1) is used for the local solver.

4.2.2 Padding with Ghost Elements

The local solver works theoretically but due to the lack of neighbour information the solver generates results that are incoherent with neighbouring blocks. As a result boundary artifacts are generated around the border of each block.

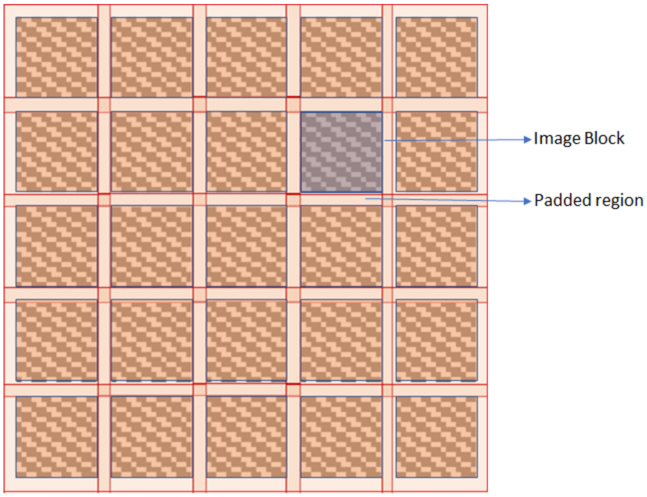


Figure 4.1: Padded Mask

One method of eliminating the artifacts is to include neighbourhood data for all blocks. To incorporate neighbourhood information, a padded region is used to offset the image block. The padded region contains ghost elements which are known pixels; the pad size is $kernelWidth/2$, where $kernelWidth$ is the width of the convolution filter. This changes the binary mask with a padding as illustrated in Figure 4.1. This resolves the issue of border artifacts by using neighbourhood information in convolution.

4.2.3 Thread and Block Configuration

CUDA performance depends mainly on thread configuration and a configuration that is a multiple of 16 produces the best performance. Various thread configurations were tested and the 16×16 thread configuration produced the best results.

The number of blocks in each dimension depends on the number of threads per block in each dimension. P will denote the padding size and the number of threads in x and y dimension will be *threadsize_x* and *threadsize_y* respectively. The method to determine the number of required blocks in x and y dimensions is as follows:

$$blockNumber_x = \frac{(imageWidth - P)}{(threadSize_x + P)}$$

and

$$blockNumber_y = \frac{(imageHeight - P)}{(threadSize_y + P)}.$$

4.2.4 Using Shared Memory

The process of accessing shared memory is very fast compared to accessing global memory which can significantly improve performance. There are constraints to using shared memory however as listed below:

- Limited amount of memory per block (usually 48 KB per block).
- Uncontrolled execution of threads may lead to race conditions where two or more threads try to read and write to the same memory location and only one gets access.
- After writing to shared memory each time $\_\_syncthreads()$ is used to maintain correct execution order. Since the solver is running locally, $\_\_syncthreads()$ does not work as a global fence. This is used to make sure all threads in a block maintain synchronization.
- The scope and access to shared memory is limited to threads that reside in a block. Threads from one block cannot access shared memory from other blocks.

At the beginning, shared memory needs to be filled up with pre-fetched data for better performance. Once the shared memory is loaded with data, computations are performed inside shared memory.

4.2.5 Shared memory initialization

The thread configuration per block is 16×16; therefore there are 256 threads per block. The shared memory size per variable is allocated for 256 floating point numbers. The first thread of each block is used to initialize the entire shared memory. Once the input data is copied from global to shared memory, it is no longer necessary to access global memory; this results in lower memory traffic and better overall performance.

4.2.6 Convolution

The FFT based convolution used in the global solver was fast but the problem is that it requires data to be copied back and forth between the CPU and GPU which became the bottleneck of the program. A tiling strategy implemented in convolution on the individual block level eliminates the copying of memory back and forth thus resulting in improved solver performance. The convolution filter values are constant and this allows the convolution filter to be stored in constant memory. another advantage is that constant memory is faster than global memory. Once the data is loaded, each individual thread calculates its own convoluted value with a filter. The algorithm for convolution is depicted in Algorithm 2. Every thread needs to execute the convolution function to calculate the convoluted value for its corresponding pixel location.

Algorithm 2 Convolution on padded block

```
**localIndex** corresponds to thread index in each block;
**temp** is an array which resides in shared memory initialized with block image data
**haloArray** is array of padded block
**MASK_H, MASK_W**: represents filter height and width respectively
$corner\_x := threadIdx.x - MASK\_W/2$;
$corner\_y := threadIdx.y - MASK\_H/2$;
for($y = 0$ to MASK_H)
{
    for($x = 0$ to MASK_W)
    {
        int $i = cornar\_x + x$;
        int $j = cornar\_y + y$;
        $maskIndex = x + y * MASK\_W$;
        if ($i < 0$ or $i >= blockDim.x$ or $j < 0$ or $j >= blockDim.y$)
        {
            $i = i + MASK\_W/2$;
            $j = j + MASK\_H/2$;
            $haloIndex = i + j * (blockDim.x + 2 * padSize)$;
            $imageValue = haloArray[holoIndex]$;
        }
        else
        {
            $imageIndex = i + j * blockDim.x$;
            $imageValue = temp[imageIndex]$;
        }
        $convolutedValue = convolutedValue \times imageValue$;
    }
}
$convolutedResult[localIndex] = convolutedValue$;
```

4.2.7 Vector Dot Product

Another important adjustment that was made to the local server is the vector dot product. To calculate the vector dot product the point wise multiplication is performed in the kernel. The result is stored in shared memory that is the same size as the number of threads of each block. Once the point wise multiplication results are derived then the addition is achieved using a parallel reduction operation [51]. Parallel reduction is a tree based approach that is used within each thread block. The idea of parallel reduction is depicted in Figure 4.2. The traditional computational complexity of summing n numbers is $O(n)$ where as the the parallel reduction complexity is only $O(\log n)$.

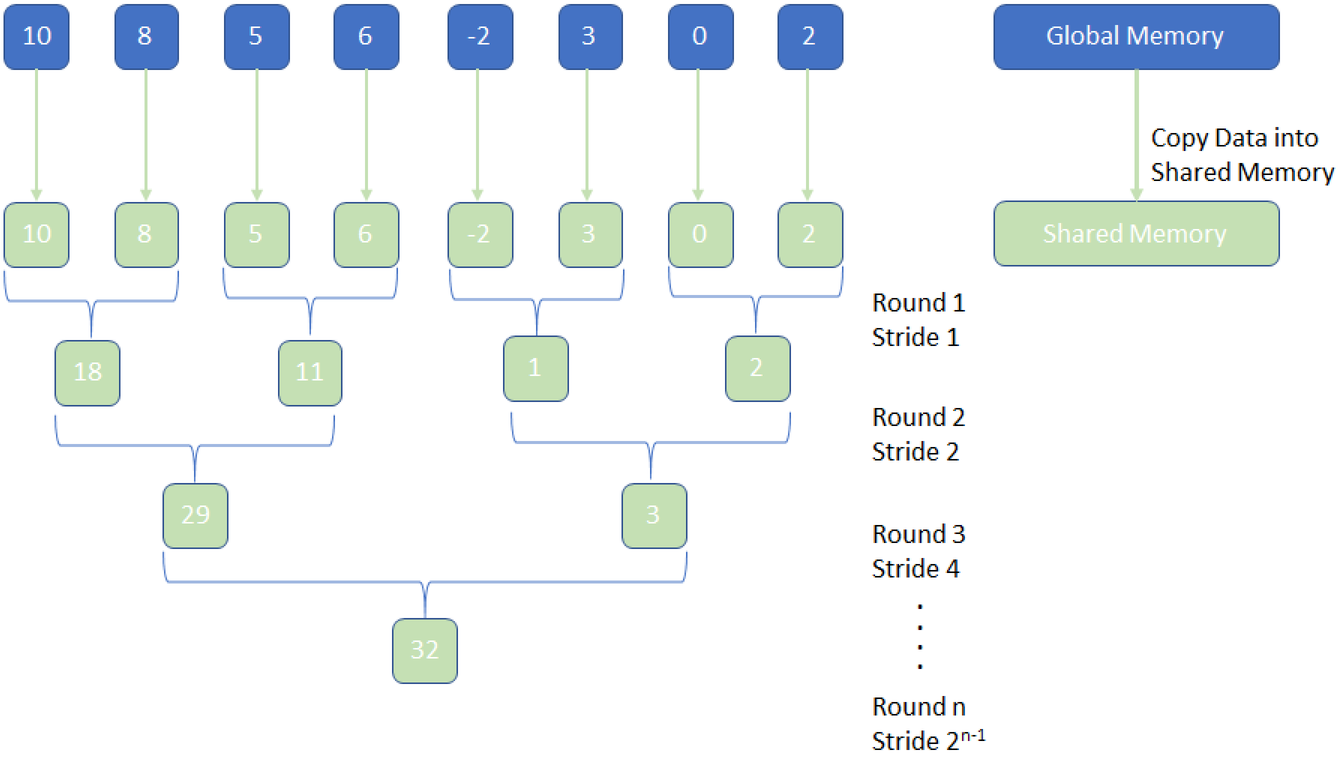


Figure 4.2: Parallel Reduction

Each individual block is assigned a final vector dot product after the reduction operation process. Since a local solver is being used, each block will have a different dot product result and different number of iterations for the convergence. Since shared memory is used, each variable will make their own copy for each individual block.

4.3 Adaptive Sampling

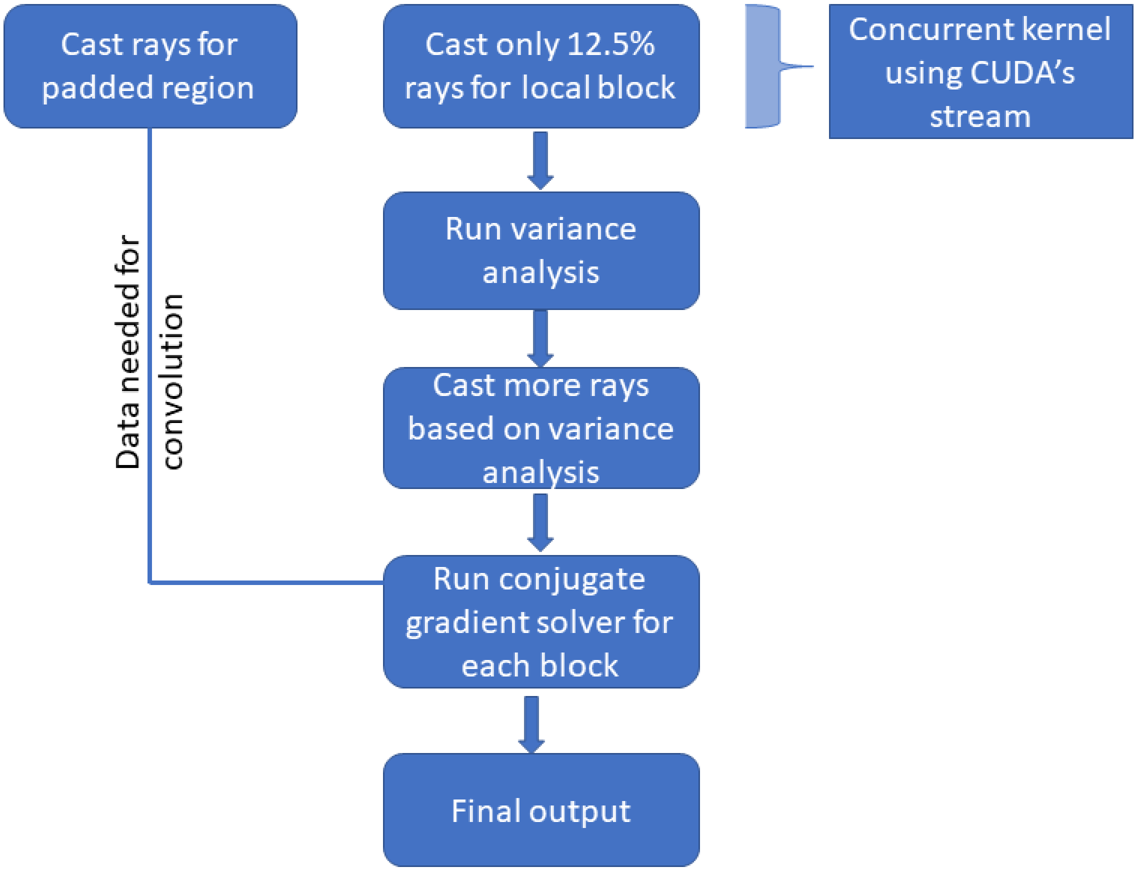

Cast rays for padded region
Cast only 12.5% rays for local block
Concurrent kernel using CUDA's stream
Data needed for convolution
Run variance analysis
Cast more rays based on variance analysis
Run conjugate gradient solver for each block
Final output

Figure 4.3: Flow chart for adaptive sampling strategy

The methodology is further enhanced with an adaptive sampling strategy. Since the solver runs locally and the sampling matrix of each block is independent of others, the adaptive sampling strategy is used to further reduce the computational cost. This technique is used initially to cast only 12.5% or 32 rays per block (in order to reduce warp divergence, any multiple of 32 threads can be used). Also, 100% of the rays on the padded region are cast simultaneously in a parallel stream. Based on the volume rendering result for each block, variance analysis is performed on the turned on pixels. This variance analysis gives the information of variance for each block; depending on the variance analysis, blocks with higher variances are given additional rays to cast. In the next step, the rays are cast for those additional pixels; when this ray casting is completed the local solver is run on each of the blocks. Then different blocks are assigned with various number of pixels using this method. The same local solver performs the reconstruction and it also uses the padded regions pixel information for the convolution part. The flow chart for adaptive sampling strategy is given in Figure 4.3. The only difference between this strategy with the local solver is that it uses two steps to perform the ray casting operation. As shown in Figure 4.3, the first step is used for casting rays in the padded region as well as in each block. Then the variance analysis is performed simultaneously to find the blocks with higher variances. The next step performs additional ray casting operations for each block with higher variances. Finally the conjugate gradient solver recovers the missing pixels to generate the final output.

Chapter 5

Results and Discussions

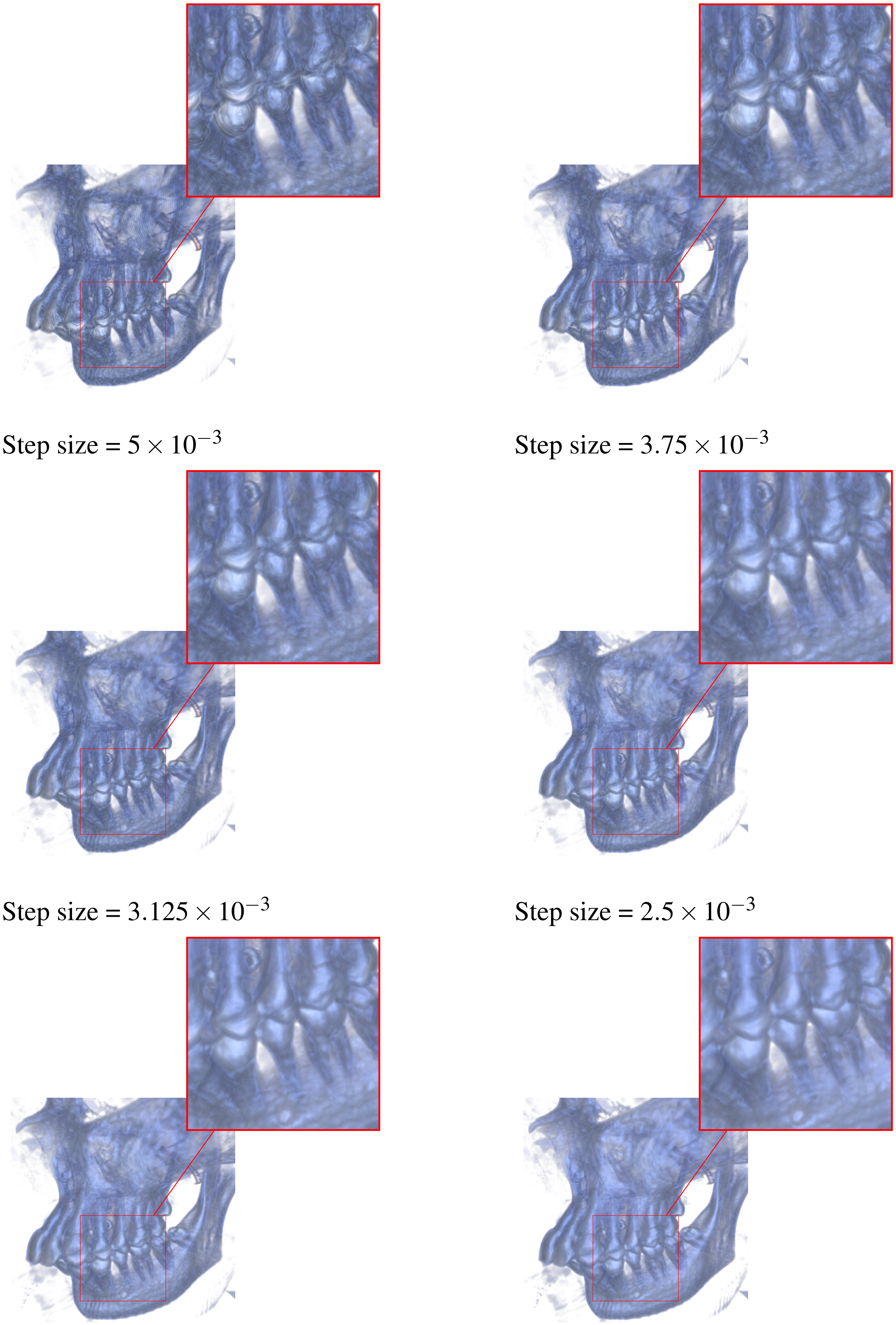
This chapter presents the experimental results and their associated analyses. The first section discusses the experimental results obtained with high quality rendering and how high quality rendering affects the performance of a volume renderer. The trade-off between high quality rendering and performance are also depicted in this chapter with visual examples. The second section talks about the results of the experiments on the compressive volume rendering; and a comparison of running a renderer locally and globally will be examined.

5.1 High quality rendering

High quality rendering is achieved using a very high sampling rate, an accurate interpolation method and realistic shading techniques. An interpolation method is needed to reconstruct a continuous scalar function for the discrete sampled data and also plays a crucial role in the quality of the volume renderer. Then the shading model contributes by generating a higher quality image. In this thesis, consideration is given to parameters for generating high quality images such as step sizes along the rays, interpolation methods and shading models.

5.1.1 Step size

The utilization of smaller step sizes always produces better quality and more details in the image. The contrary is true for high step sizes as information may be lost stepping through the volume. The phenomenon is depicted in Figure 5.1. The top left figure in 5.1 uses the largest step size and the most bottom right figure utilizes the lowest step size. The difference between the two images is apparent; and the smaller step size also generates the smoother image.



Step size = 5×10^{-3}

Step size = 3.75×10^{-3}

Step size = 3.125×10^{-3}

Step size = 2.5×10^{-3}

Step size = 1.875×10^{-3}

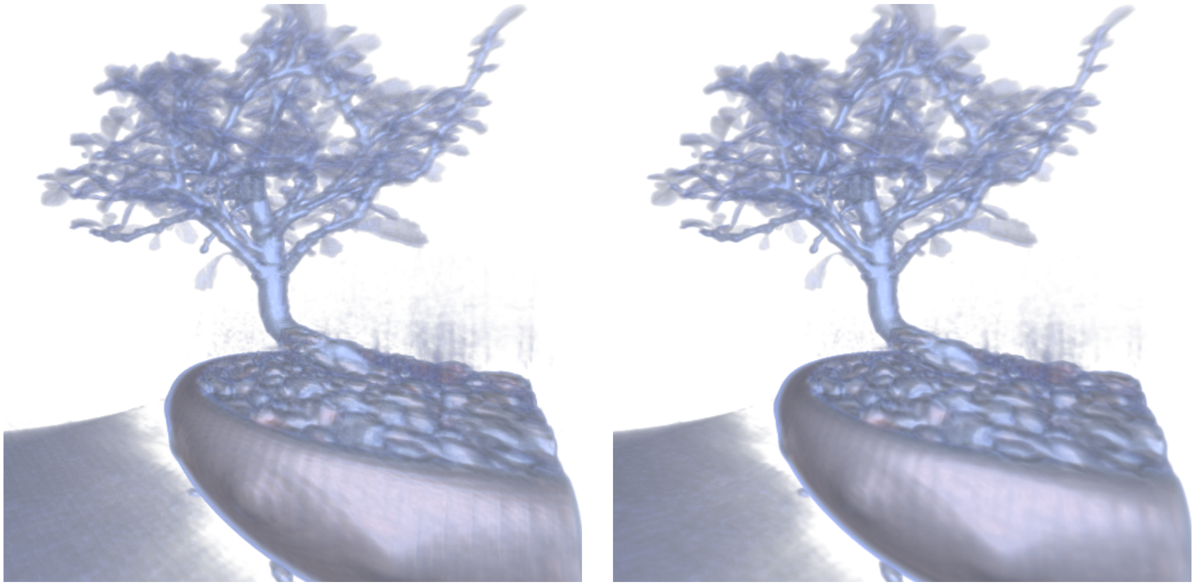
Step size = 1.25×10^{-3}

Figure 5.1: Image quality for different step sizes

5.1.2 Interpolation method

The image quality is also significantly affected by the method of interpolation used. This thesis looks at two different interpolation methods for the volume renderer to use in reconstructing an image. The two techniques examined are tri-linear interpolation and tri-cubic interpolation. A main differentiating characteristic of the two interpolation methods is the number of neighbour information to approximate a point. The tri-linear method uses 8 neighbour points while the tri-cubic technique uses 64 points of information to approximate a value for any location.

A volume renderer using the tri-linear and tri-cubic interpolation techniques are exhibited in Figure 5.2. It can be noted that the generated image using tri-cubic interpolation is smoother than the image generated using tri-linear interpolation.



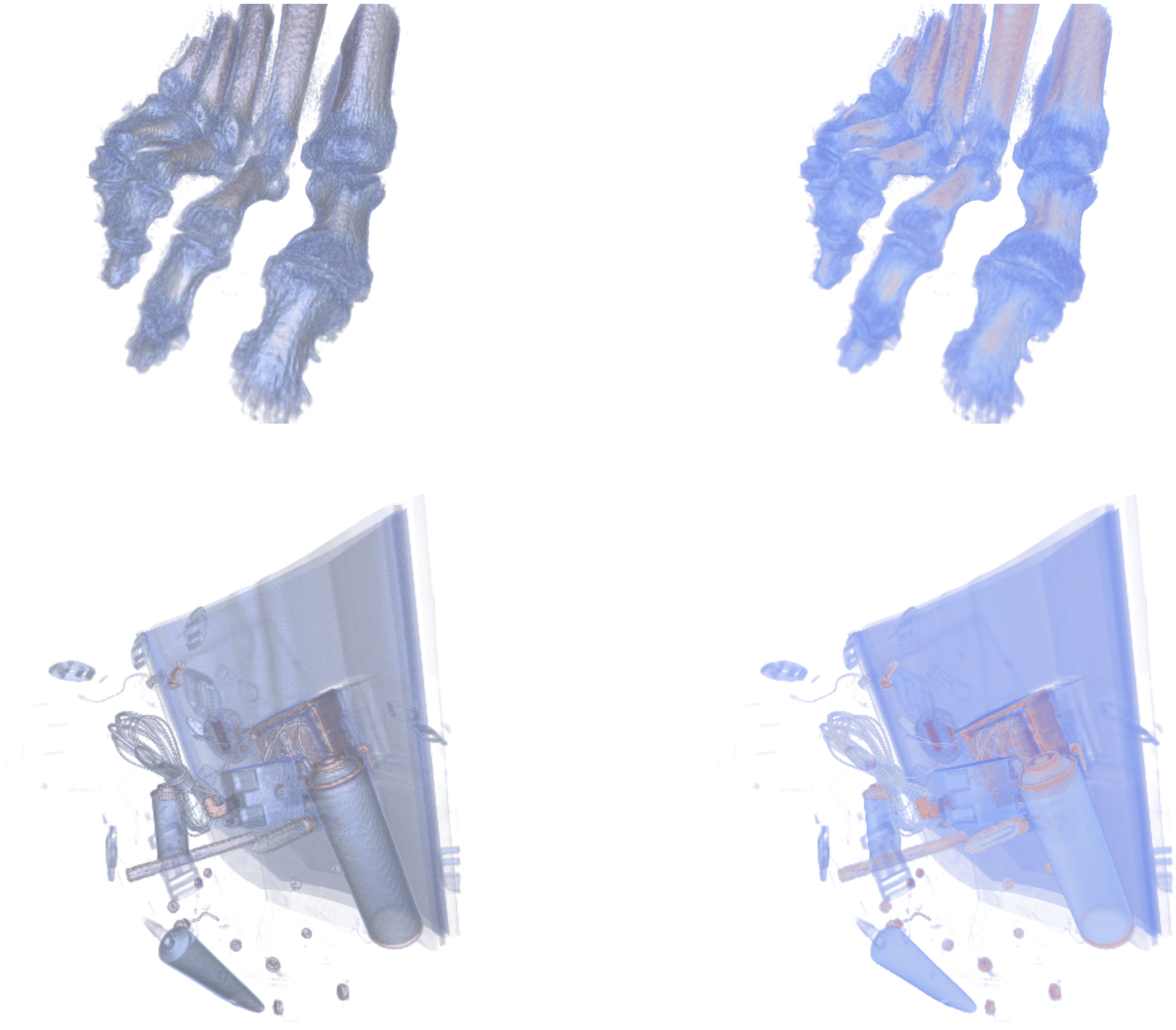
Tri-linear interpolation

Tri-cubic interpolation

Figure 5.2: Comparison of images generated using three different interpolation methods

5.1.3 Shading model

The shading model that is chosen also affects the rendered image and can improve the image significantly. This thesis shows a comparison of the rendered image based on a lighting versus no lighting environment. The shading technique that is used in this thesis is a method proposed by Phong et al. [9]. Figure 5.3 shows this comparison of an image with lighting and no lighting. Using a very minute gradient step size it is possible to create a surface on a rendered volume. Consideration is given to the 2^{nd} degree derivatives in each direction for lighting at every point; as a result texture memory is accessed 6 times (twice for each direction).



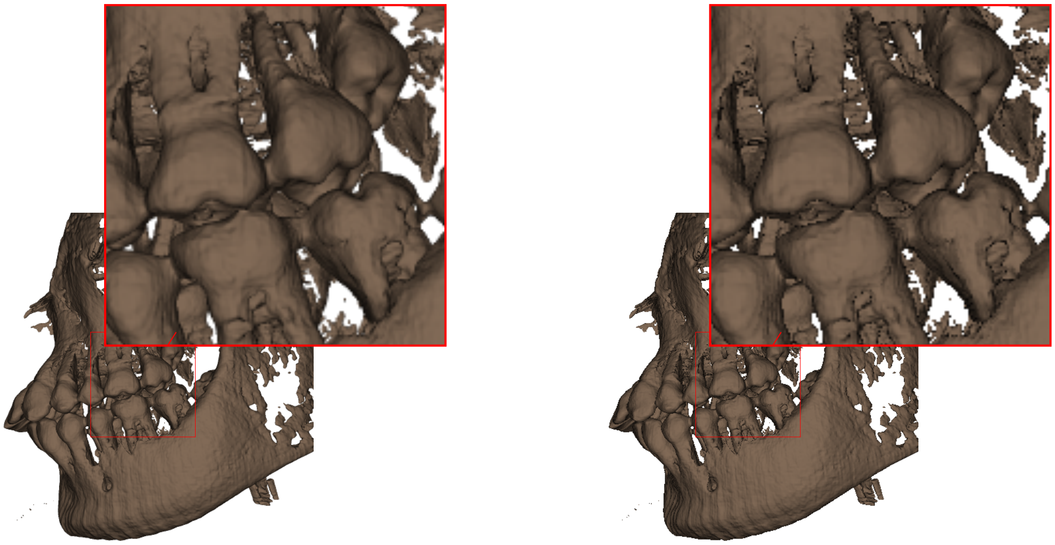
With shading

Without shading

Figure 5.3: Comparison of images while turning on light vs without lighting

5.1.4 Super sampling

The use of super sampling can also generate high quality rendered images. This thesis uses 5 samples per pixel and then divides the pixel into 4 parts. Then rays are cast from the center of each part and at the end rays are also cast from the center of each pixel. Finally the average is calculated to finalize the value for that particular pixel. In Figure 5.4, a comparison of an image using 5 samples per pixel and 1 sample per pixel is shown. Super sampling significantly improves the image quality by minimizing aliasing effects.

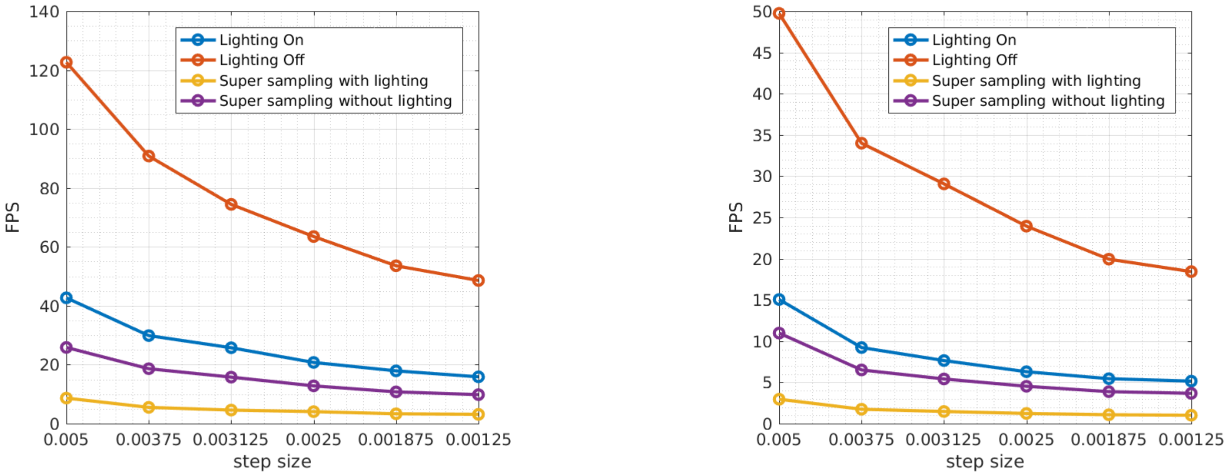


Super sampling (5 rays/pixel) 1 rays/pixel

Figure 5.4: Super sampling vs regular sampling

5.2 Quality vs Performance

In the previous section a discussion of parameters for high quality rendering took place; there are other techniques that can also generate high quality rendering such as global illumination, multiple light sources, and etc.. There is an inverse relationship between the quality and performance in a volume renderer; and a high quality renderer never guarantees interactivity. Figure 5.5 shows how step size affects the performance of both tri-linear and tri-cubic interpolation. Generally using tri-linear interpolation is faster but the more features that are added the frame per second (FPS) drops significantly with the decreasing step size. Figure 5.5 shows a comparison of frame rates for tri-linear and tri-cubic interpolation with shading. Then by using super sampling the aliasing effect is eliminated in the rendered image. Figure 5.5 also compares the performance of tri-linear and tri-cubic interpolation with sampling rate of 5 rays per pixel. All these figures exemplify the need for improvement in the performance of the renderer and one way to increase the performance is with compressive rendering.



Tri-linear interpolation

Tri-cubic interpolation

Figure 5.5: Changes of FPS for different interpolation methods

5.3 Compressive Rendering

Compressive rendering is defined as the process of reconstructing a full image by using only a subset of the pixels. The main contribution of this thesis is the reconstruction process using a gradient solver. Liu and Alim et al. [4] shows the reconstruction time a renderer takes using a CPU. This thesis proposes a parallel based approach of the reconstruction and the solution has two approaches. In the first approach the image is reconstructed globally with the solver running the entire image. In the second approach the image is partitioned into blocks; then the blocks are run through the solver and merged to finalize the output image.

5.3.1 Global solver

In this thesis there was only one solution for the entire image in the global solver; and the solver can be affected by one parameter. The performance of the solver is affected by the number of iterations; and the number of iterations can be fixed or a flag can be set to allow the solver to run multiple times until it arrives at a desired solution.

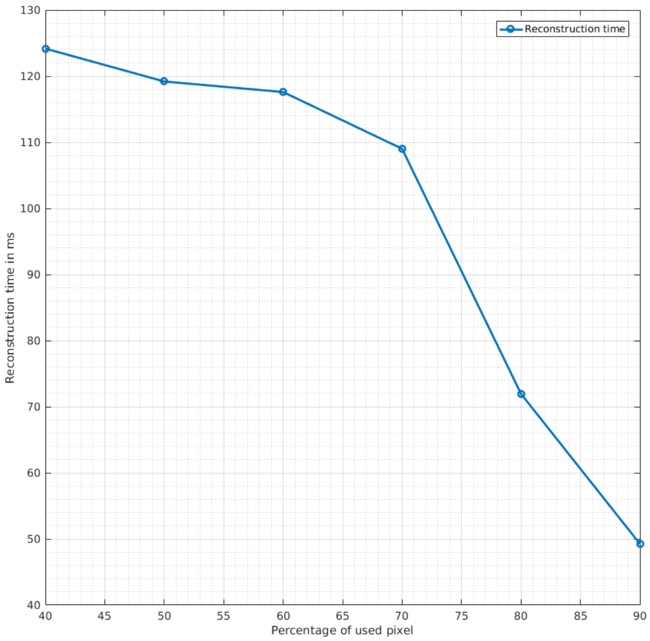


Figure 5.6: Reconstruction time of global solver for various missing pixel

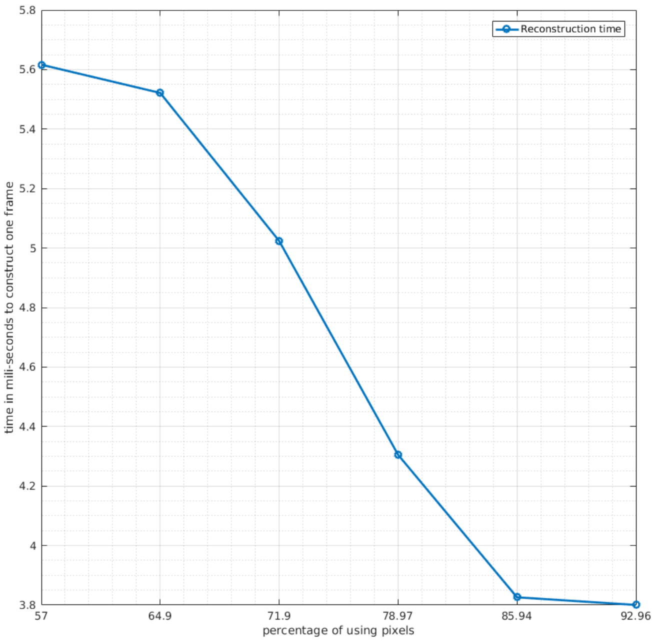


Figure 5.7: Reconstruction time of local solver for various missing pixel

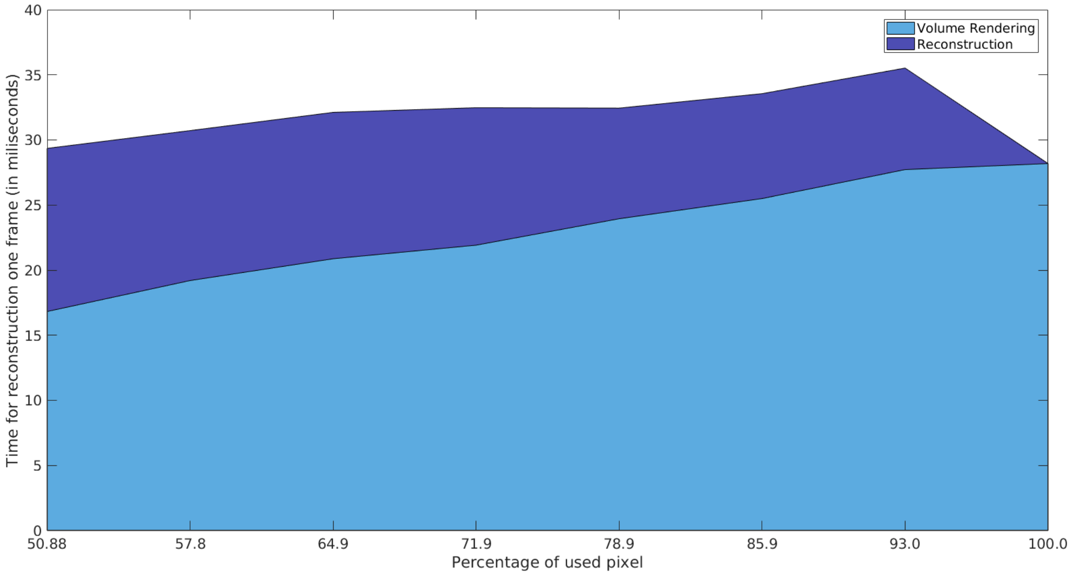
The most time consuming operation in our solver is the convolution process; by using FFT based convolution memory is copied back and forth from CPU to GPU for each iteration. This copying process between CPU and GPU results in a very slow reconstruction time. Figure 5.6 shows the reconstruction time of a frame for various missing pixels. As an example, if 60% of the pixels are missing or 40% of the pixels are present the time it takes to reconstruct a frame is approximately 124 milliseconds. If this is converted to a frame per second calculation; the reconstruction frame rate will be about 10 frames per second which is not well suited for interactivity. Therefore a new strategy had to be proposed to increase the frame rate reconstruction process to accommodate interactivity.

5.3.2 Local solver

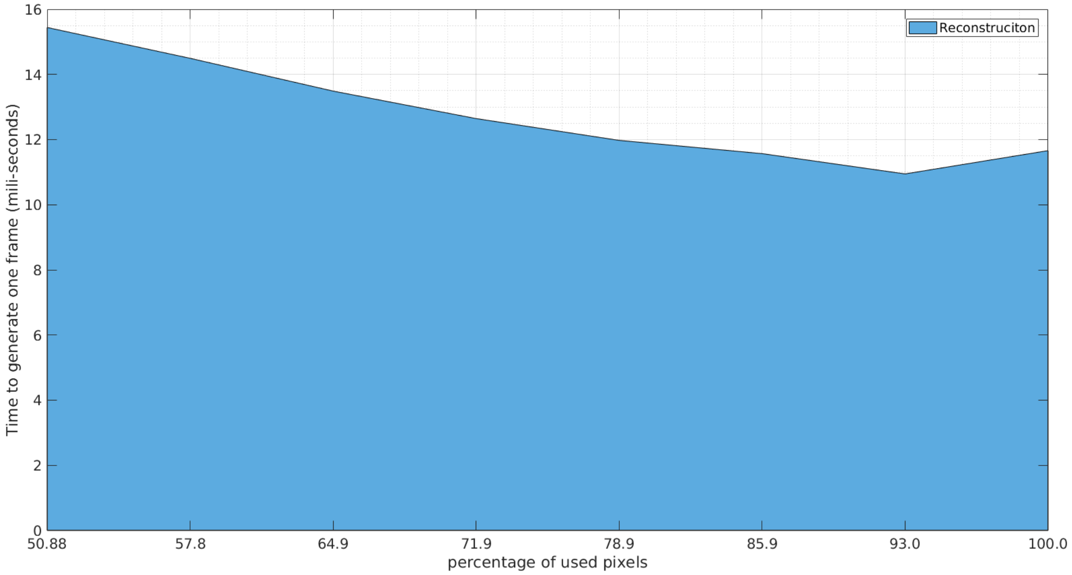
One technique to optimize the performance of the solver is to partition the problem into multiple blocks and then run the solver locally on the block level. In this process the solver can leverage

CUDA's blocking scheme and shared memory to avoid expensive movement of data. This blocking with tiling strategy is discussed in Chapter 4; using the tiling strategy each block will have its' own solution. Another benefit of tiling is that it reduces the matrix size significantly resulting in a faster reconstruction rate. This phenomenon is exemplified in Figure 5.7. Comparing figure 5.6 and 5.7 it is easily to distinguish the performance of the two solvers. For example, using only 60% pixels, the global solver takes 119 milliseconds where the local server only takes 5.6 milliseconds to process. In terms of performance the local solver is 24 times faster than the global solver.

The following results prove that the conjugate gradient solver saves on computational cost and time. To test the performance of the solver more features are added to the volume render and rendering time measurements are taken so comparisons can be made. In this section there are 8 diagrams plotting the time it takes for the renderer and solver to run with the additional features. The first four diagrams deal with tri-linear interpolation method and the final four deal with the tri-cubic interpolation method. The diagrams also show each interpolation method without shading, with shading, with super sampling without shading and finally super sampling including shading. Each figure contains two diagrams; the top figure contains the time required to construct one frame by volume renderer and conjugate solver. The bottom figure shows the overall FPS for that particular feature. In each diagram the percentage of used pixels is plotted on the x axis and the time it takes to calculate/render is on the y axis. The above figures make it evident that the more features and pixels added to the volume renderer the more time it takes where as the reconstruction time is almost constant. When comparing the total rendering time; it is obvious that volume rendering using 100% of the pixels takes more time than the compressive rendering method. Therefore using the conjugate gradient solver the rendering time can be significantly reduced.

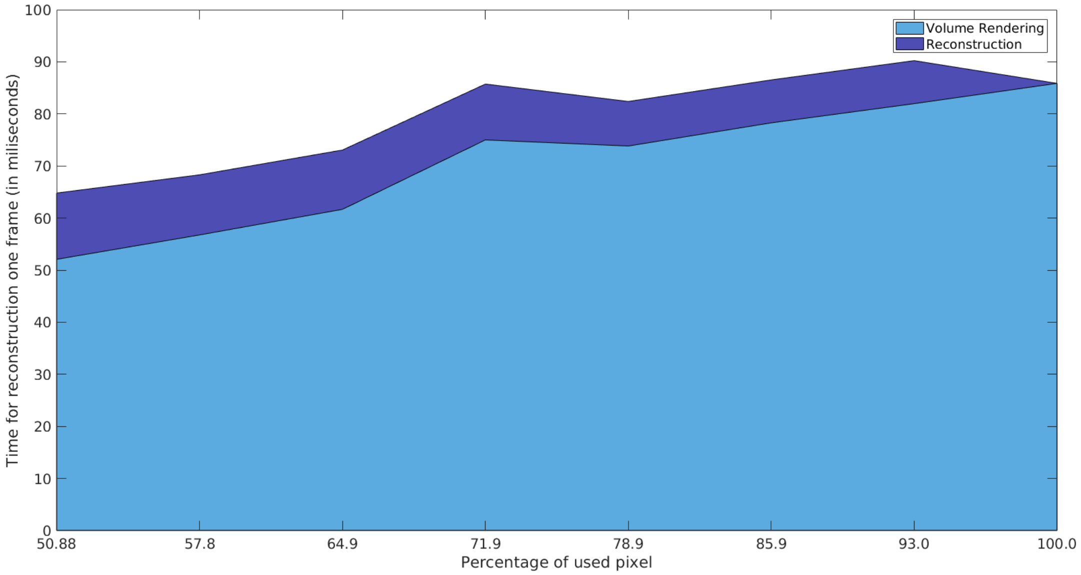


Tri-linear interpolation without shading

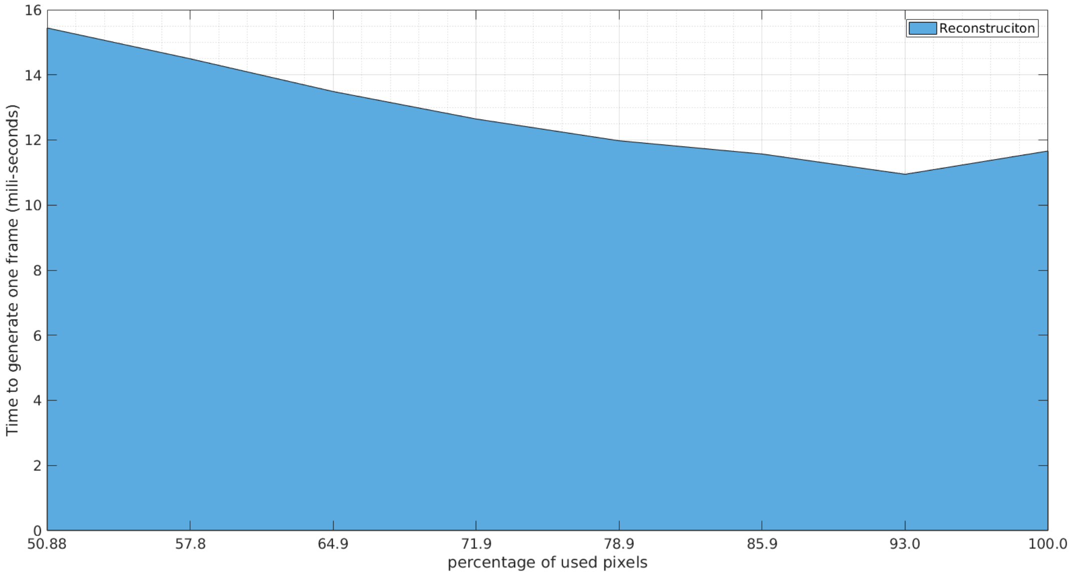


Over all FPS

Figure 5.8: Timing diagram for tri-linear interpolation without shading

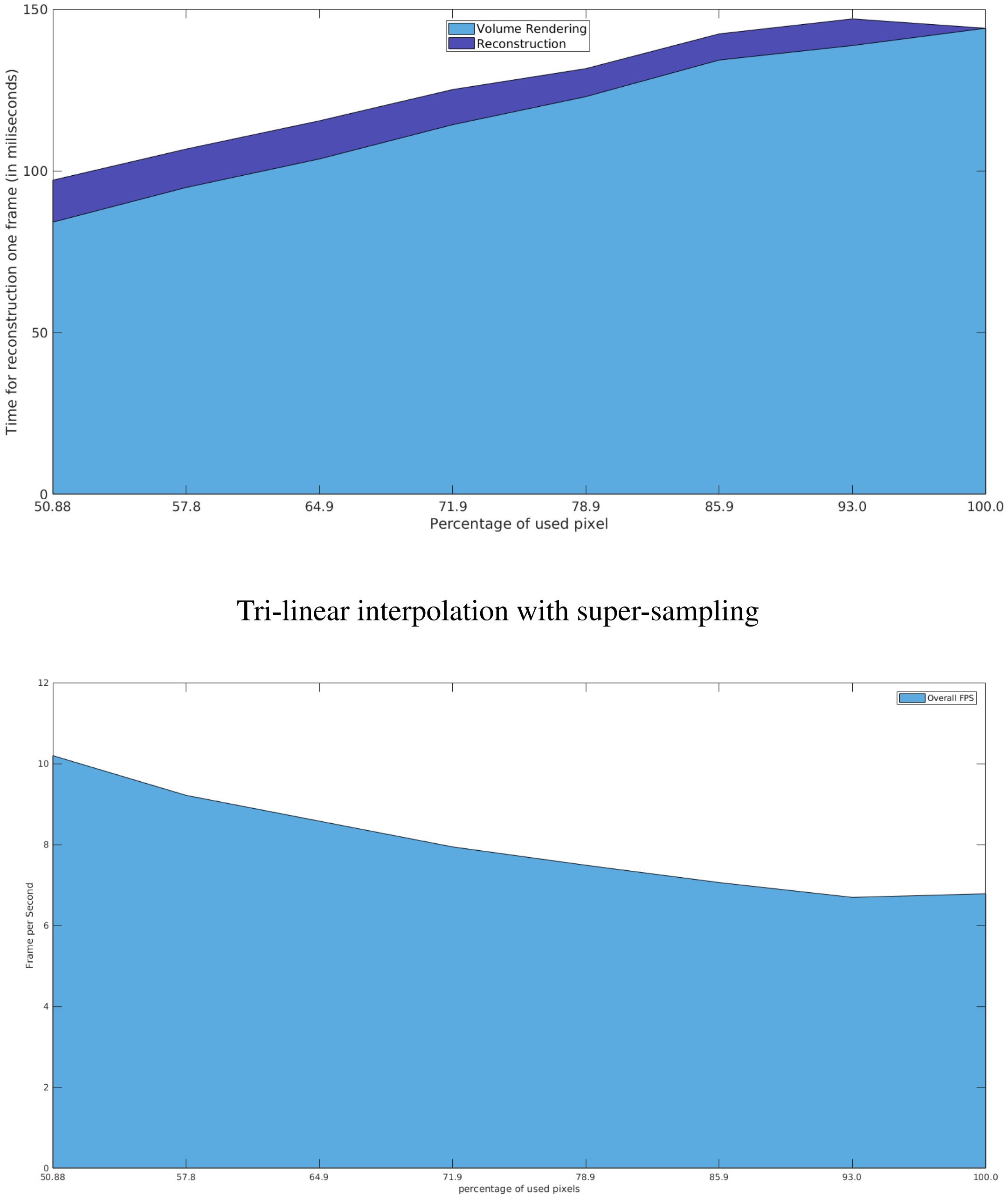


Tri-linear interpolation with shading



Over all FPS

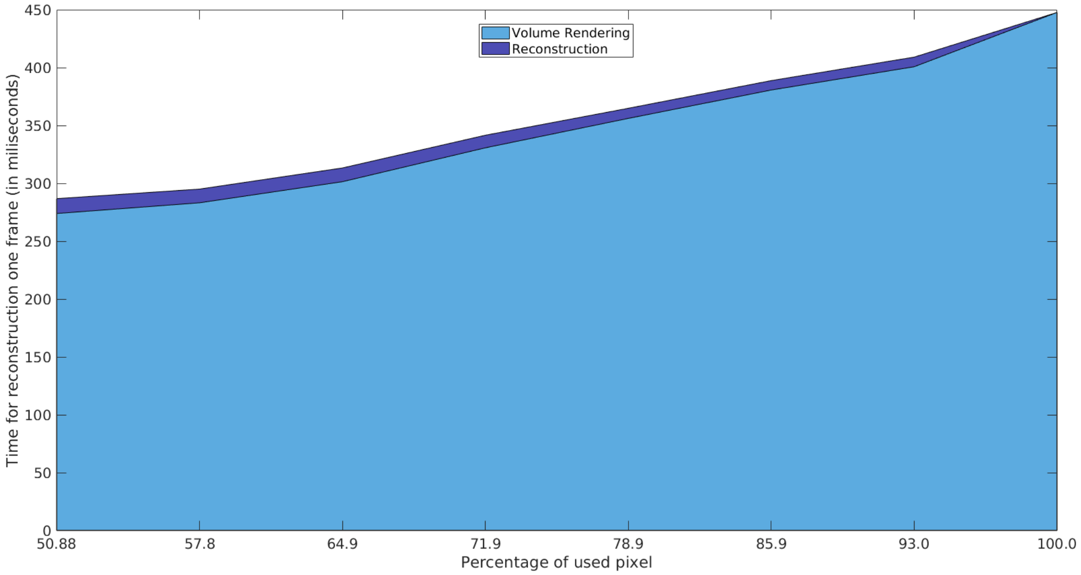
Figure 5.9: Timing diagram for tri-linear interpolation with shading



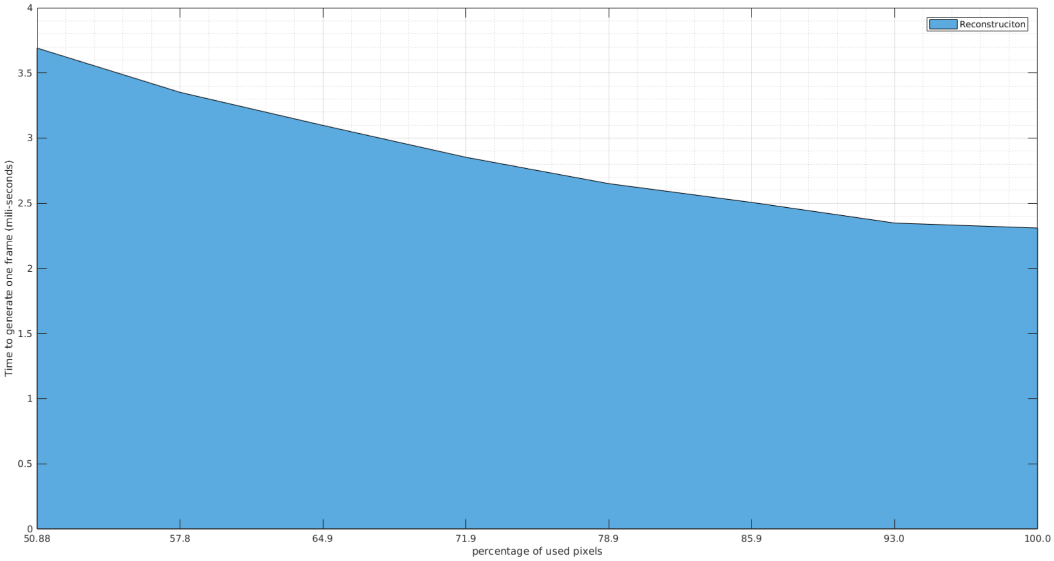
Tri-linear interpolation with super-sampling

Over all FPS

Figure 5.10: Timing diagram for tri-linear interpolation with super-sampling

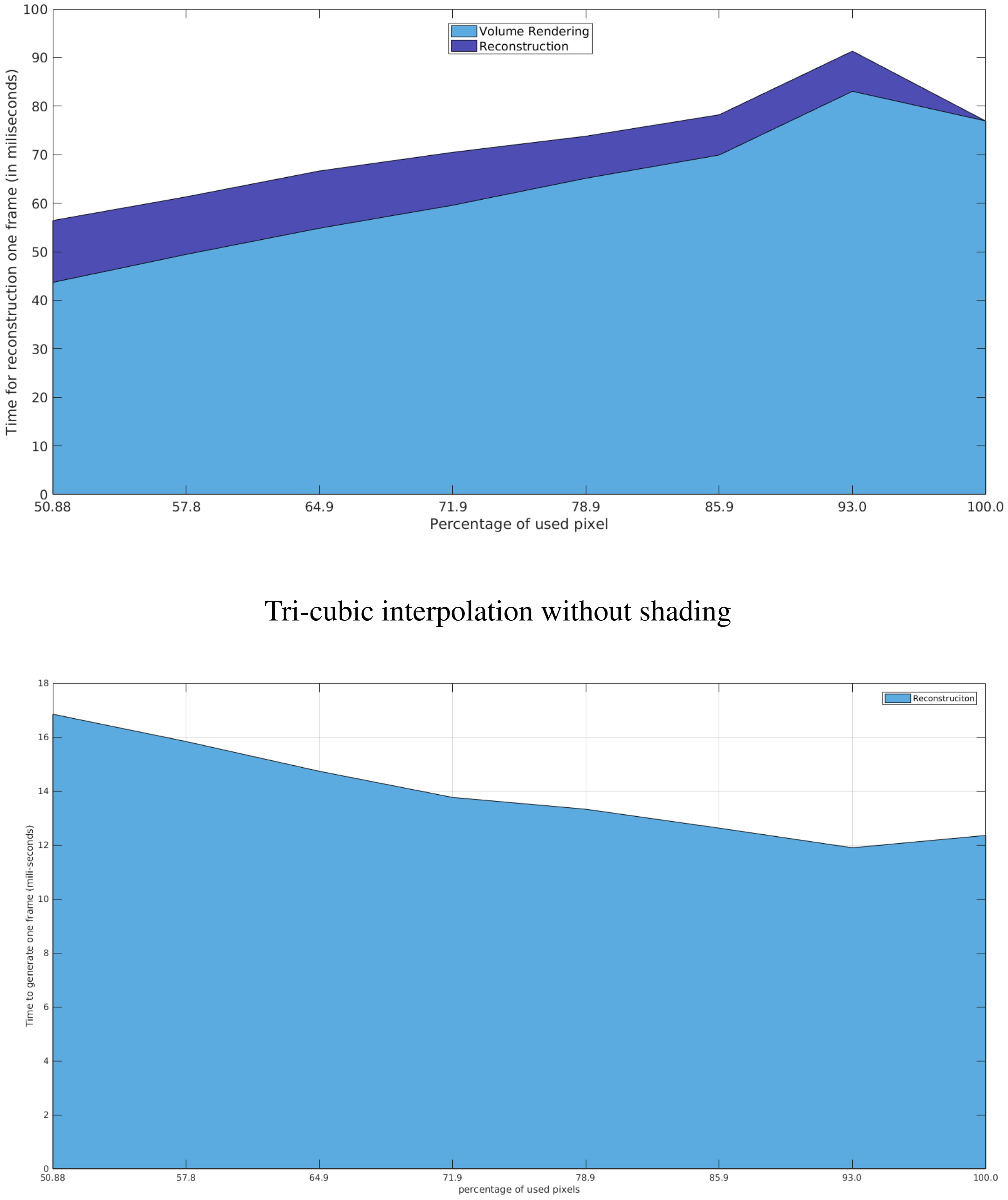


Tri-linear interpolation with super-sampling and lighting



Over all FPS

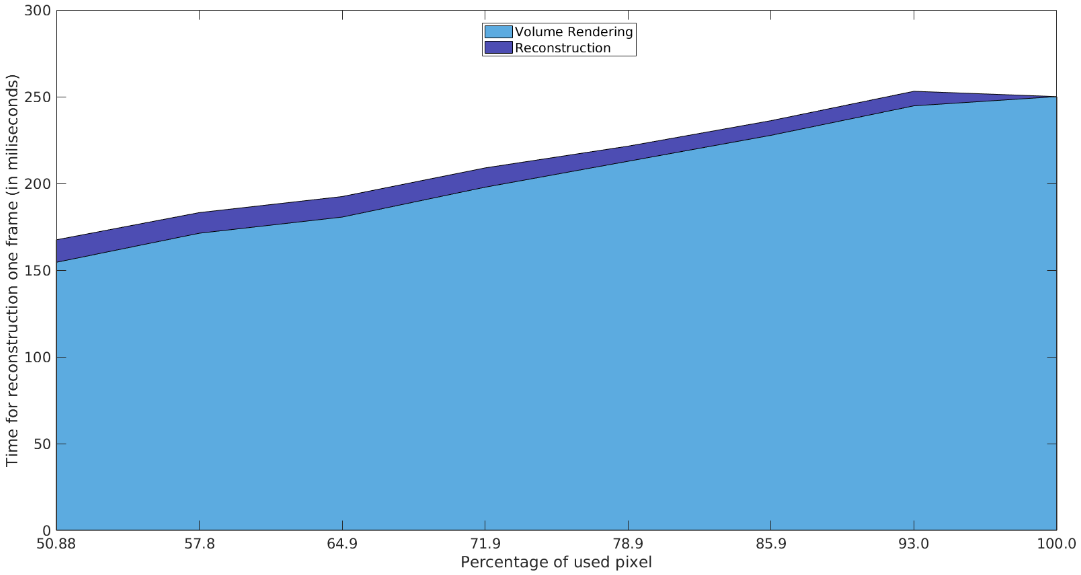
Figure 5.11: Timing diagram for tri-linear interpolation with super-sampling and shading



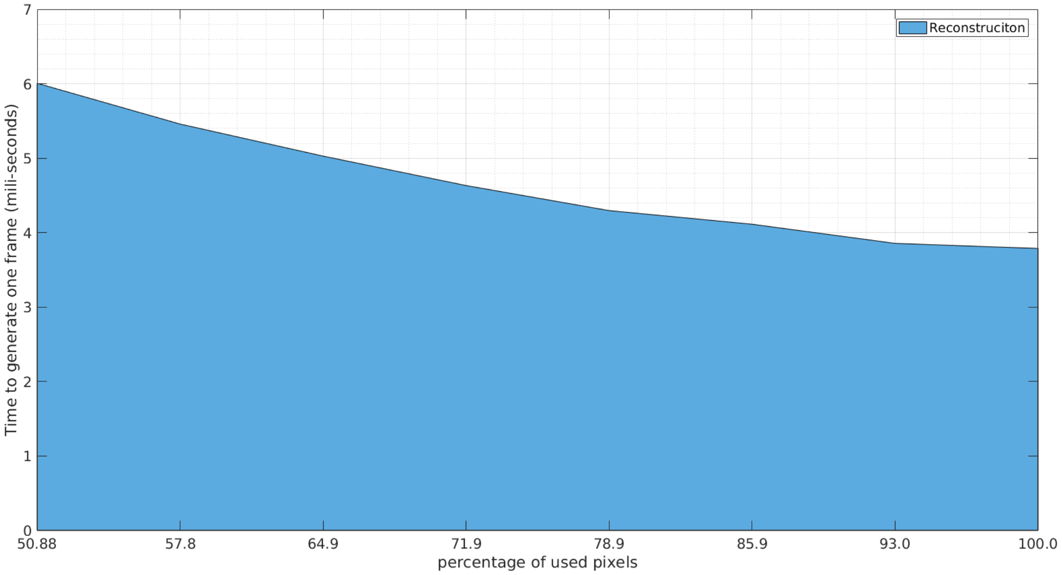
Tri-cubic interpolation without shading

Over all FPS

Figure 5.12: Timing diagram for tri-cubic interpolation without shading

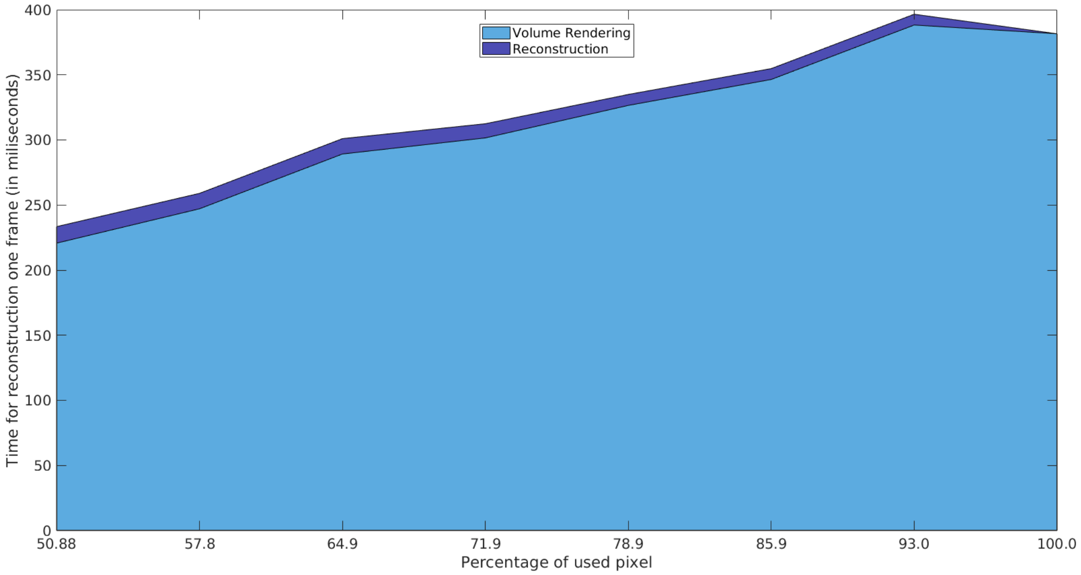


Tri-cubic interpolation with shading

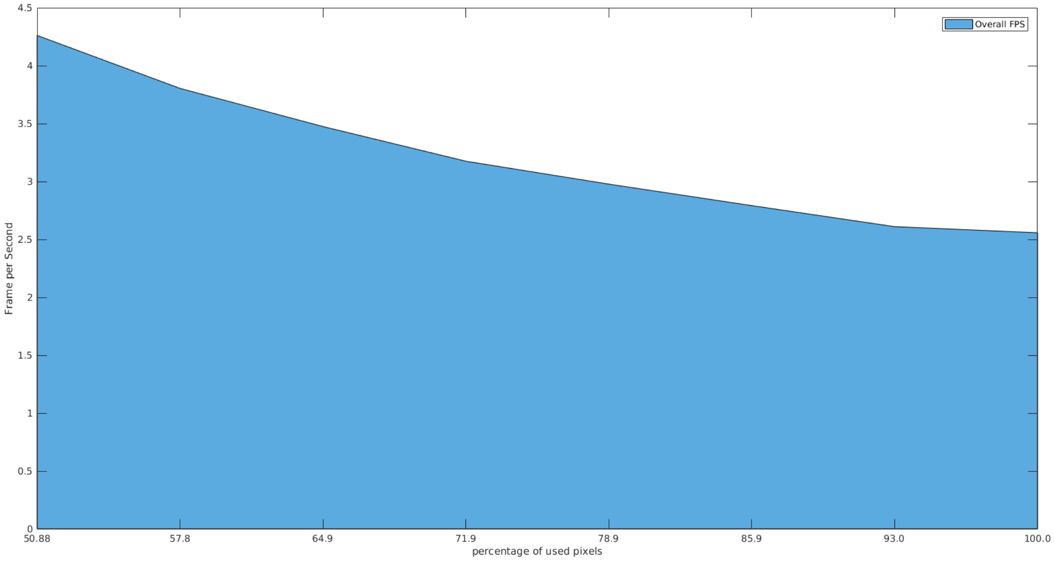


Over all FPS

Figure 5.13: Timing diagram for tri-cubic interpolation with shading

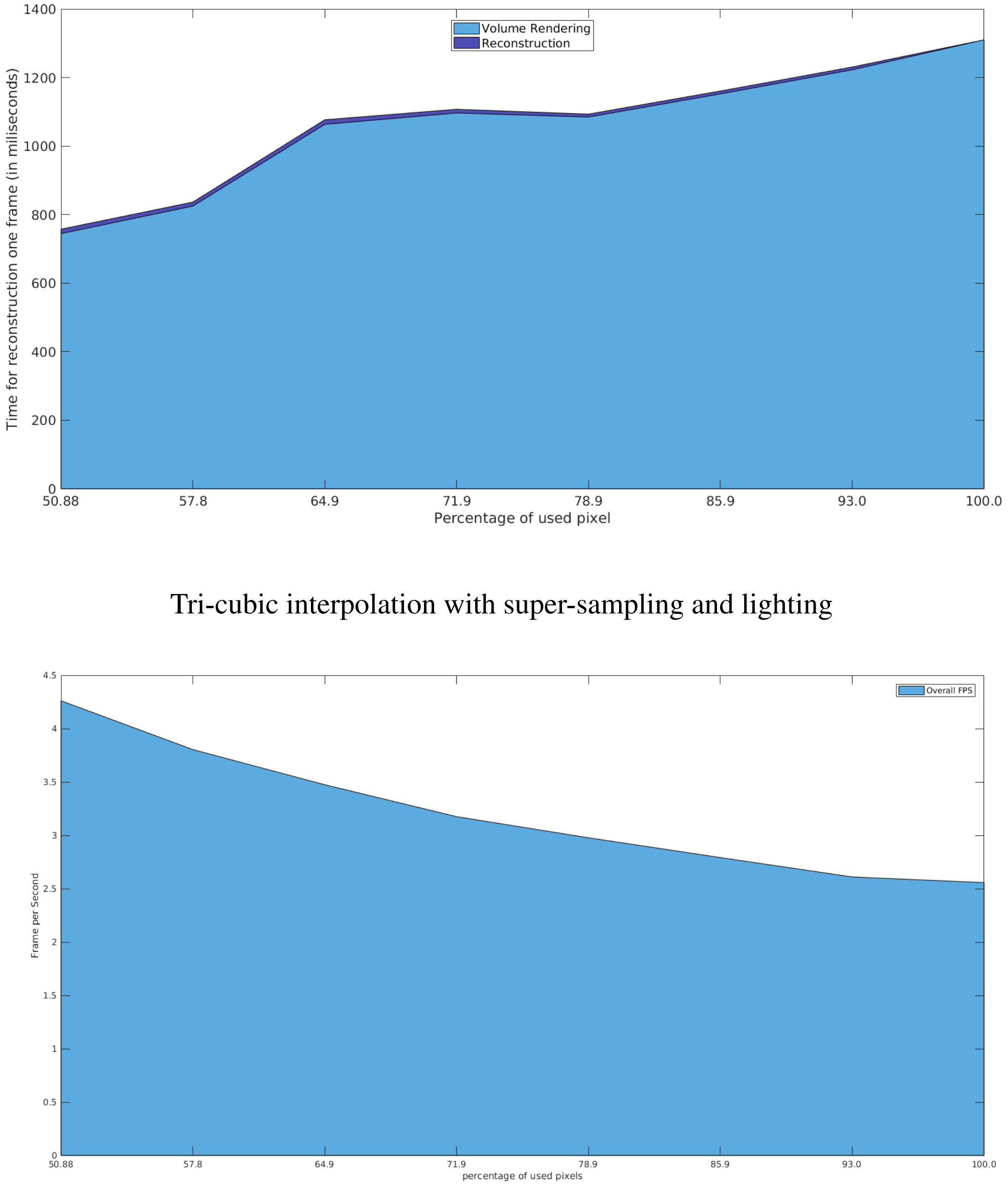


Tri-cubic interpolation with super-sampling



Over all FPS

Figure 5.14: Timing diagram for tri-cubic interpolation with super-sampling



Tri-cubic interpolation with super-sampling and lighting

Over all FPS

Figure 5.15: Timing diagram for tri-cubic interpolation with super-sampling and shading

From the above figures, it is obvious that the more features and more amount of pixels if we add to our volume renderer the more time it takes where as the reconstruction time is almost constant. If we compare the result of total rendering time in each figure, we can see that volume rendering using 100% pixel always takes more time than the compressive volume rendering. We can significantly reduce total volume rendering time by using our conjugate gradient solver.

5.3.3 Quality of local solver

This thesis focuses on the two attributes: performance of the renderer and the quality of the rendered image. The calculation of PSNR shows how the reconstruction method converged after the block level reconstruction. Figure 5.16 shows the PSNR for linear interpolation method with various conditions such as shading vs. no shading, super sampling with shading and etc.. Figure 5.17 shows the PSNR for tri-cubic interpolation with shading and super sampling.

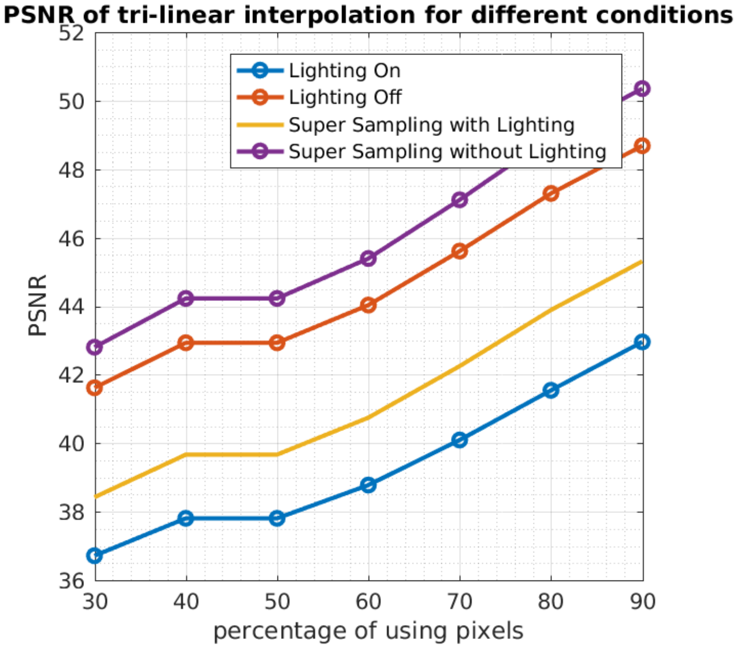


Figure 5.16: PSNR for different missing pixels

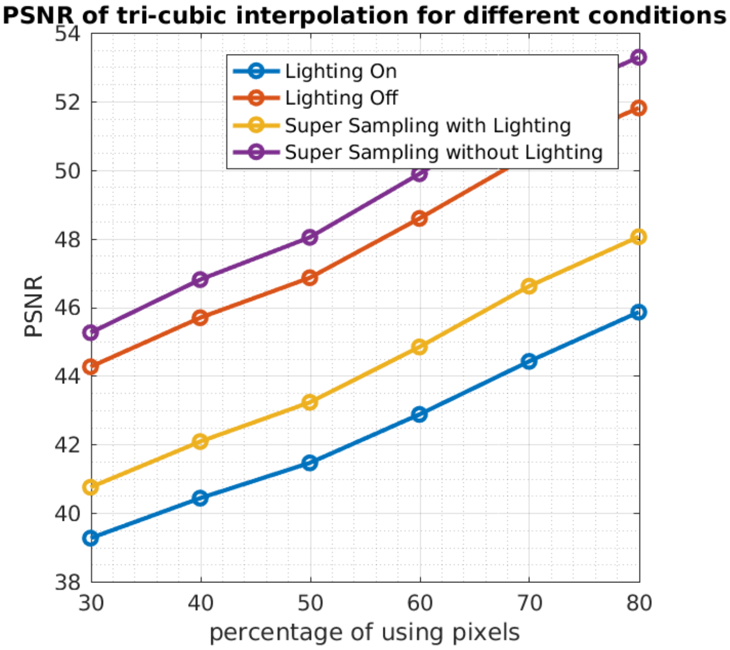


Figure 5.17: PSNR for different missing pixels

We have also shown the PSNR for iso-surface rendering in figure 5.18.

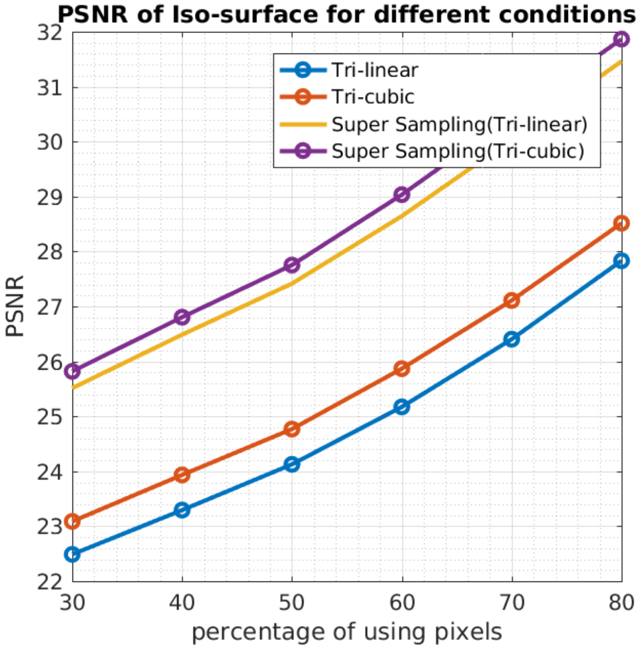


Figure 5.18: PSNR for different missing pixels for iso-surface rendering

5.3.4 Comparison of Global and Local solvers

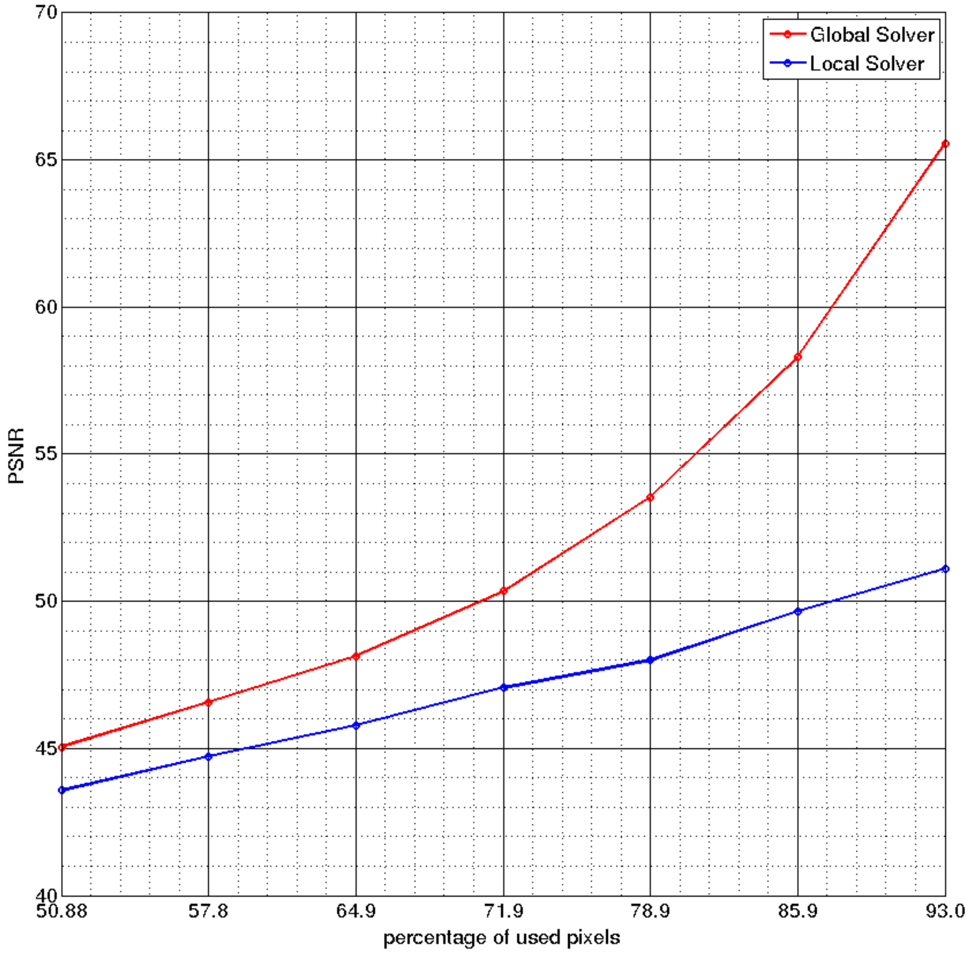


Figure 5.19: Comparison between global and local solvers for the foot image shown in Fig. 5.3 (top-left).

The local solver is selected to achieve high performance and it is being run locally for each block. It is to be noted that, the same problem is being looked at while the methodology remains the same but our implementation strategy is different. The different implementation strategy achieves high performance but there is a slight degradation in the quality of the renderer. This phenomenon is depicted in Figure 5.19. The PSNR for different percentages of used pixels are depicted where the red line represents the global solver and the blue line represents the local solver. Due to the decrease in the amount of pixels used there is a slight difference in the PSNR for both solvers. The amount of used pixels increases the PSNR for the global solver dramatically, whereas the PSNR for the local solver increases moderately. We speculate that using larger block sizes will decrease this gap but will adversely affect the performance of the local solver.

5.4 Adaptive Sampling

This thesis also further explores the methodology for adaptive sampling. Initially only 12.5% of the pixels of each block are cast; then the variance are calculated for each block. Based on the variance analysis additional pixels are added for particular blocks. Figure 5.20 represents the tri-linear interpolation method and Figure 5.21 represents the tri-cubic interpolation method. The first two bars in each figure represent the time taken by the volume renderer to render an image for 100% of the pixels. The next 2 stacked bars show the amount of time it takes for the compressive volume renderer using the adaptive sampling strategy to render an image. If we convert the total time taken by the volume renderer and the solver into FPS, it will be 12.5 and 5.75 for adaptive sampling with tri-linear interpolation and adaptive sampling with tri-cubic interpolation respectively. On the other hand, FPS for the local solver with tri-linear interpolation and tri-cubic interpolation are roughly 14 and 6 respectively. The difference in FPS between this two strategies is due to the volume rendering. Since the volume rendering part in adaptive sampling is composed of two sequential steps, this slows down the entire performance of the adaptive sampling strategy. This phenomenon can also be observed from the CUDA profiler which is shown in Figure 5.25.

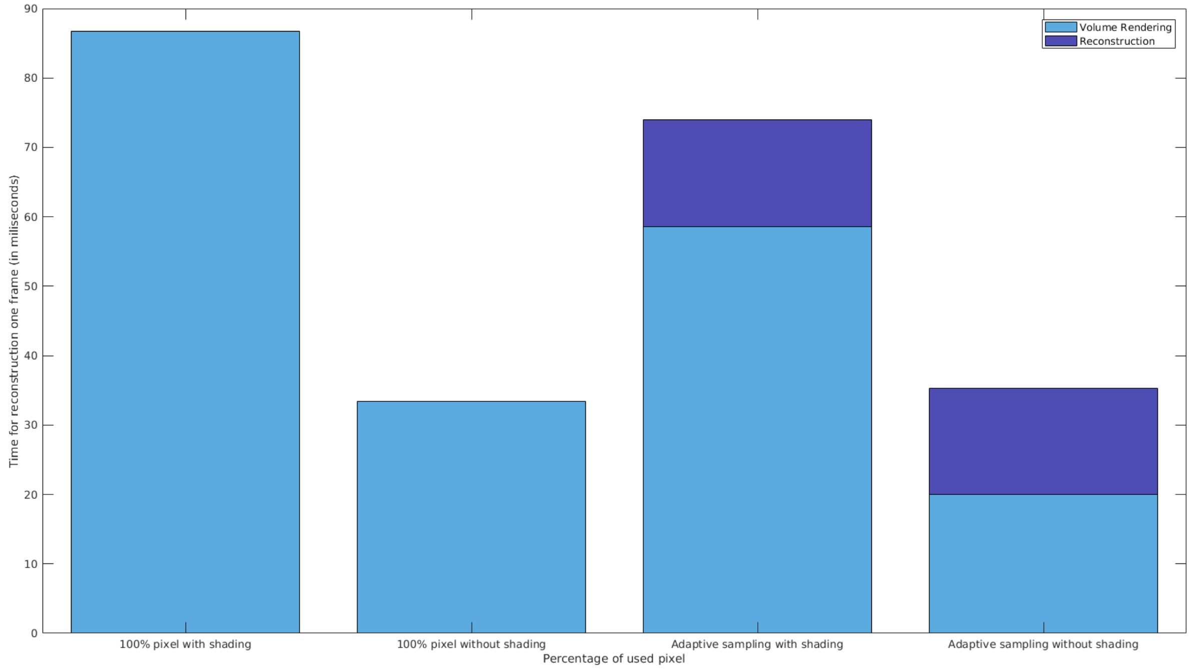


Figure 5.20: Timing diagram for adaptive sampling with tri-linear interpolation method

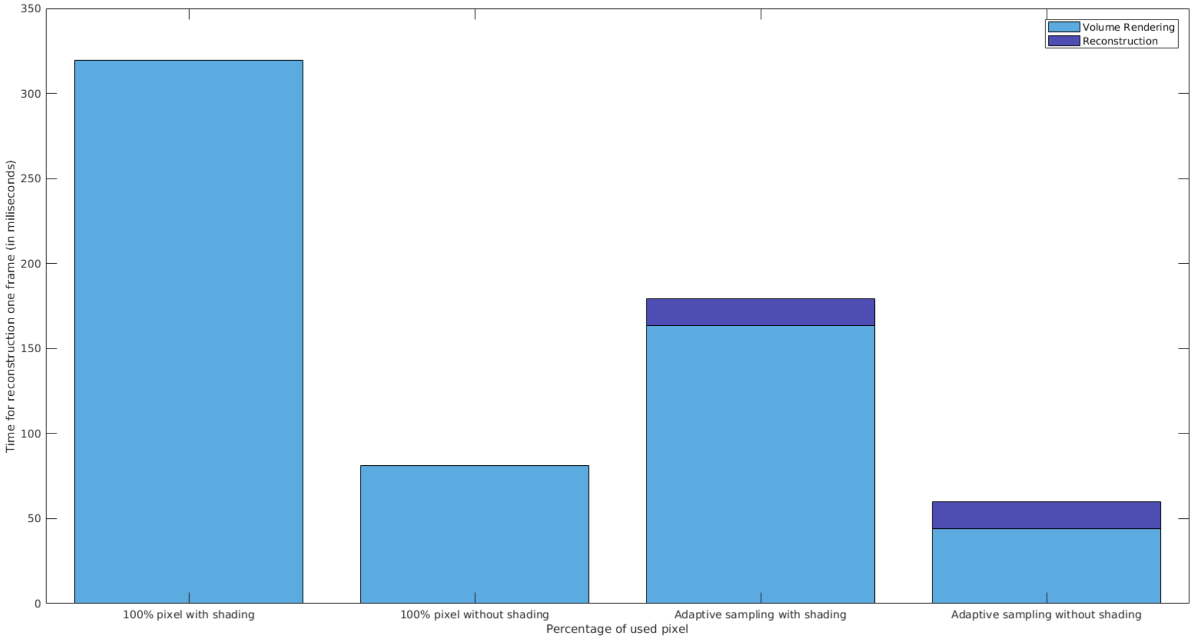


Figure 5.21: Timing diagram for adaptive sampling with tri-cubic interpolation method

5.5 CUDA profiling

5.5.1 Local Solver

For performance measurement, the CUDA profiler is used to measure the occupancy of the threads; and the overall load on the GPU for each individual CUDA kernal. In figure 5.22 it shows that a 95.5% load in the GPU is allocated for the volume renderer kernal; whereas the reconstruction kernal has a 3.6% load on the GPU. The individual kernal performance is depicted in figure 5.23 and 5.24 for renderer and reconstruction kernal respectively. In figure 5.24 the thread occupancy rate for the reconstruction kernal is 62.5% which is efficient for a CUDA based application. It is not possible to get to a 100% occupancy rate because of the use of shared memory in the GPU. GPUs have a limited amount of shared memory and it is allocated to all blocks; and each block can use up to 48kb of shared memory.

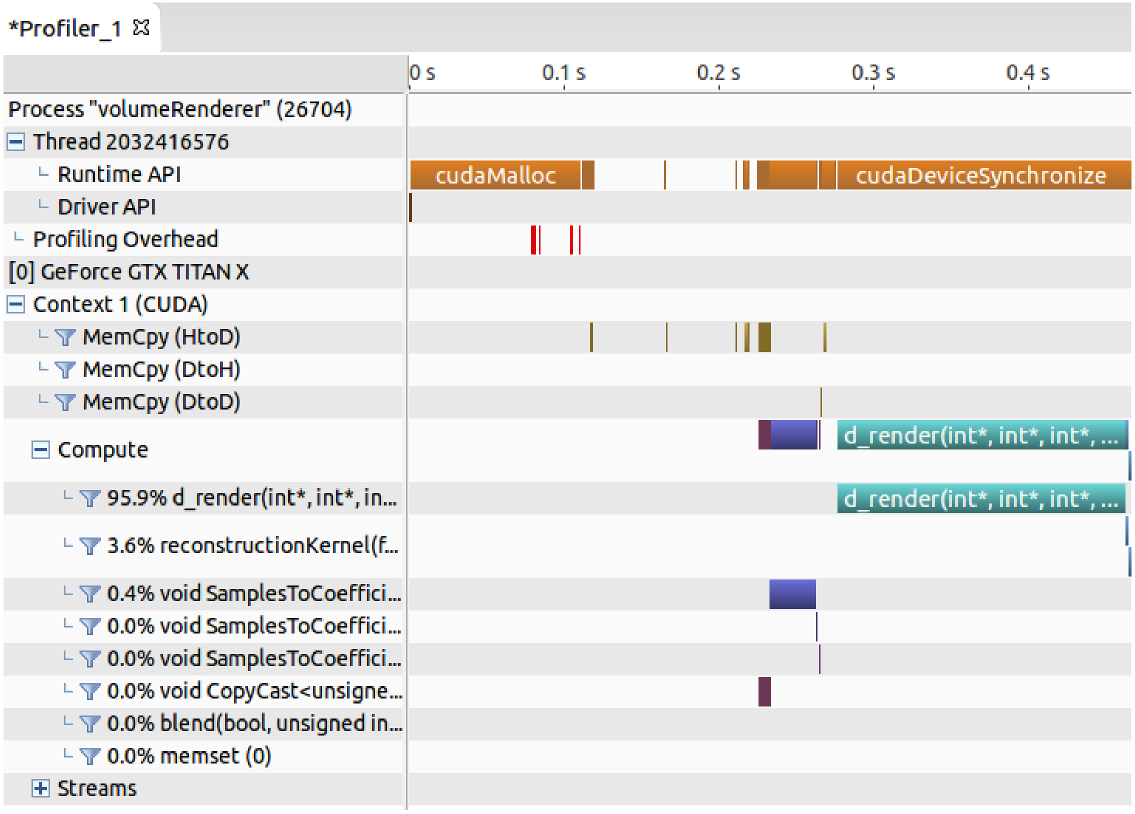
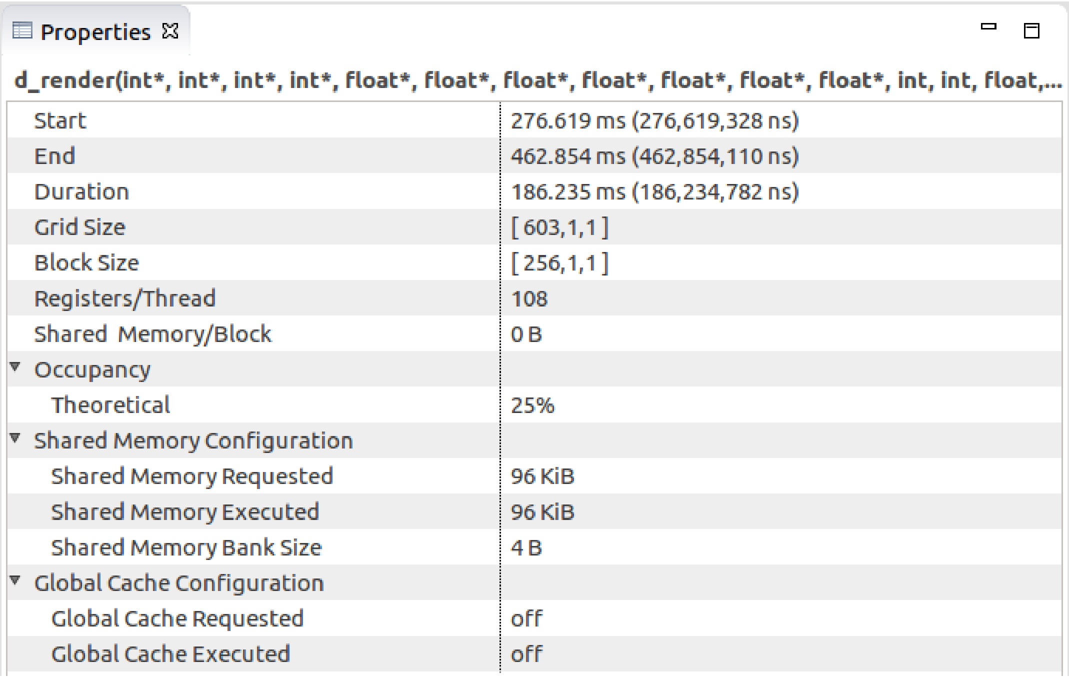

*Profiler_1
0 s
0.1 s
0.2 s
0.3 s
0.4 s
Process "volumeRenderer" (26704)
Thread 2032416576
Runtime API
cudaMalloc
cudaDeviceSynchronize
Driver API
Profiling Overhead
[0] GeForce GTX TITAN X
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
d_render(int*, int*, int*, ...
95.9% d_render(int*, int*, in...
d_render(int*, int*, int*, ...
3.6% reconstructionKernel(f...
0.4% void SamplesToCoeffici...
0.0% void SamplesToCoeffici...
0.0% void SamplesToCoeffici...
0.0% void CopyCast<unsigne...
0.0% blend(bool, unsigned in...
0.0% memset (0)
Streams

Figure 5.22: CUDA visual profiler for local solver

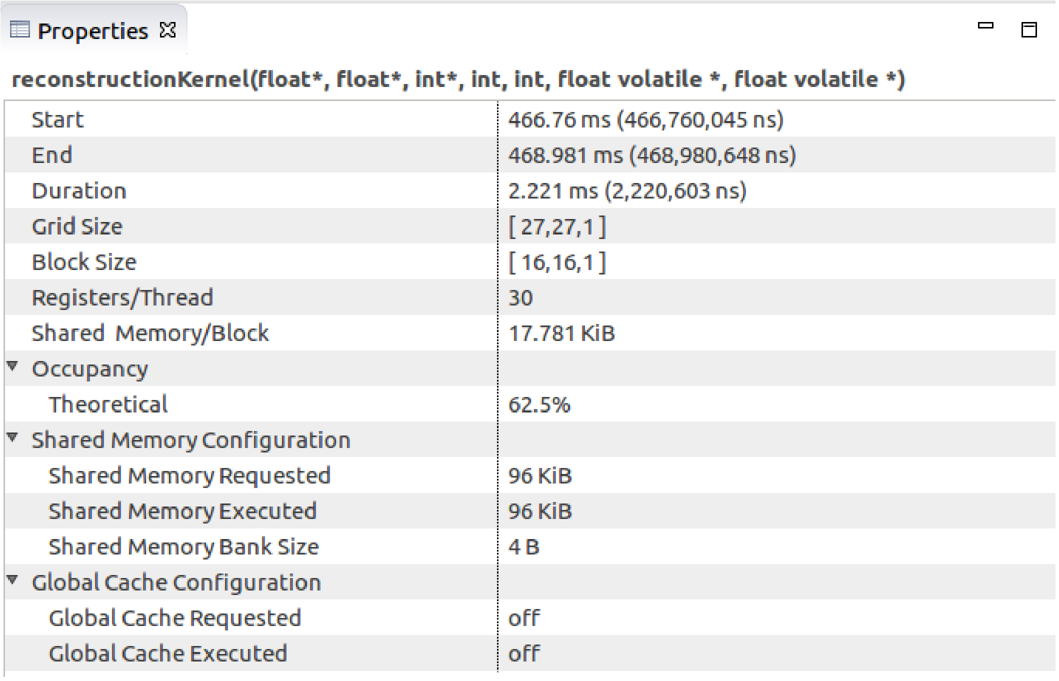


Properties

d_render(int*, int*, int*, int*, float*, float*, float*, float*, float*, float*, float*, int, int, float,...

Start	276.619 ms (276,619,328 ns)
End	462.854 ms (462,854,110 ns)
Duration	186.235 ms (186,234,782 ns)
Grid Size	[603,1,1]
Block Size	[256,1,1]
Registers/Thread	108
Shared Memory/Block	0 B
Occupancy	
Theoretical	25%
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

Figure 5.23: Volume renderer kernel description



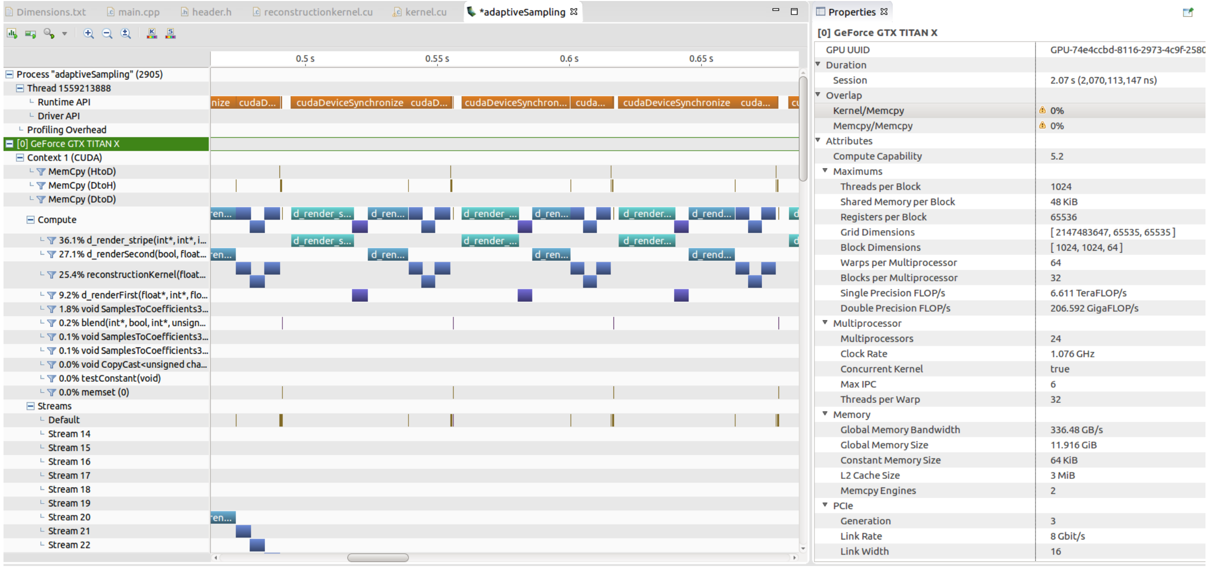
Properties

reconstructionKernel(float*, float*, int*, int, int, float volatile *, float volatile *)

Start	466.76 ms (466,760,045 ns)
End	468.981 ms (468,980,648 ns)
Duration	2.221 ms (2,220,603 ns)
Grid Size	[27,27,1]
Block Size	[16,16,1]
Registers/Thread	30
Shared Memory/Block	17.781 KiB
Occupancy	
Theoretical	62.5%
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

Figure 5.24: Reconstruction kernel description

5.5.2 Adaptive Sampling



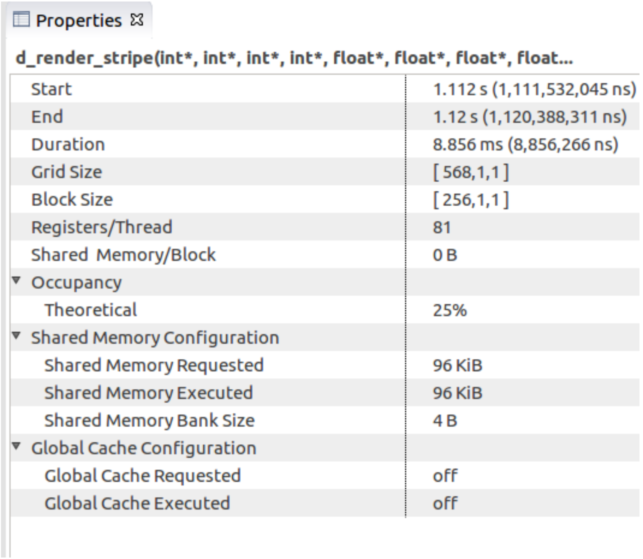
[0] GeForce GTX TITAN X

GPU UUID	GPU-74e4ccbd-8116-2973-4c9f-2580
▼ Duration	
Session	2.07 s (2,070,113,147 ns)
▼ Overlap	
Kernel/Memcpy	⚠ 0%
Memcpy/Memcpy	⚠ 0%
▼ Attributes	
Compute Capability	5.2
▼ Maximums	
Threads per Block	1024
Shared Memory per Block	48 KiB
Registers per Block	65536
Grid Dimensions	[2147483647, 65535, 65535]
Block Dimensions	[1024, 1024, 64]
Warps per Multiprocessor	64
Blocks per Multiprocessor	32
Single Precision FLOP/s	6.611 TeraFLOP/s
Double Precision FLOP/s	206.592 GigaFLOP/s
▼ Multiprocessor	
Multiprocessors	24
Clock Rate	1.076 GHz
Concurrent Kernel	true
Max IPC	6
Threads per Warp	32
▼ Memory	
Global Memory Bandwidth	336.48 GB/s
Global Memory Size	11.916 GiB
Constant Memory Size	64 KiB
L2 Cache Size	3 MiB
Memcpy Engines	2
▼ PCIe	
Generation	3
Link Rate	8 Gbit/s
Link Width	16

Figure 5.25: CUDA visual profiler for adaptive sampling

In the adaptive sampling strategy the use of CUDA's concurrent streaming processor requires that rays are cast for the padded region and for each block at the beginning. In this stage for each block only 12.5% of the total pixels residing in each block are cast. While the variable of 12.5% can be tuned; 12.5% is chosen to guarantee warp convergence. Then using 12.5% of the pixel values, calculations of the variance are made and more pixels are added based on the variance analysis for performing further ray casting operations. Once this stage is done the solver is run for the rendered subset of the pixels and it recovers all the missing pixels. To experiment on how the methodology works, the CUDA profiler is used. The profile generated for adaptive sampling is shown in Figure 5.25 and examinations of each individual kernel using a profiler is executed. In Figure 5.25, it is obvious that, the entire volume rendering operation is executing in serial not in parallel. At the first step 100% rays are cast on the padded region and 12.5% rays are cast on each block and in the next step, based on the variance analysis, additional ray casting operations are performed for the additional pixels. These sequential operations slow down the performance for the entire adaptive sampling strategy. The details of the kernel that had rays cast for the padded region is depicted in

Figure 5.26. The kernel that casts only 12.5% of rays for each block is depicted in Figure 5.27. The kernel that casts additional rays is profiled in Figure 5.28. Finally the detail profiling of the conjugate gradient solver is depicted in Figure 5.29.

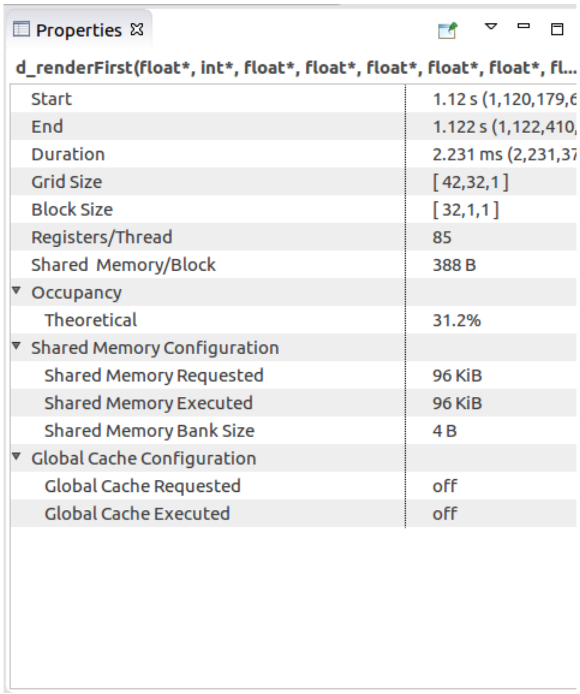


Properties ⊠

d_render_stripe(int*, int*, int*, int*, float*, float*, float*, float...

Start	1.112 s (1,111,532,045 ns)
End	1.12 s (1,120,388,311 ns)
Duration	8.856 ms (8,856,266 ns)
Grid Size	[568,1,1]
Block Size	[256,1,1]
Registers/Thread	81
Shared Memory/Block	0 B
▾ Occupancy	
Theoretical	25%
▾ Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
▾ Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

Figure 5.26: The profiler for the kernel that casted rays for padded region

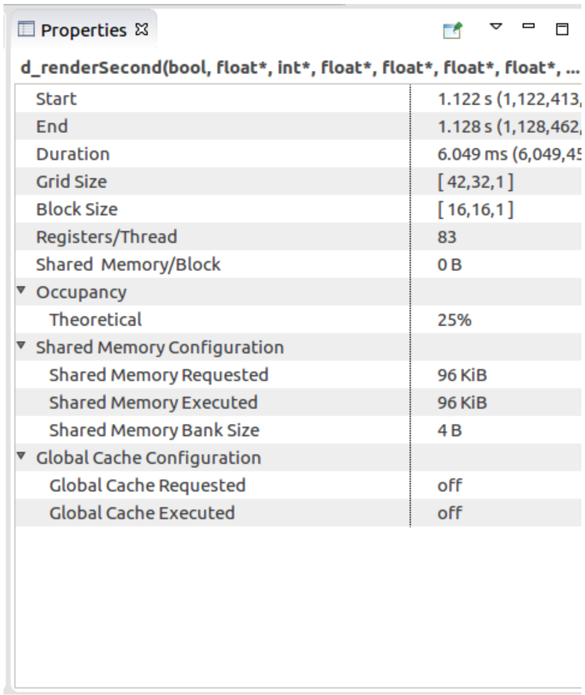


Properties ⊠

d_renderFirst(float*, int*, float*, float*, float*, float*, float*, fl...

Start	1.12 s (1,120,179,6
End	1.122 s (1,122,410,
Duration	2.231 ms (2,231,37
Grid Size	[42,32,1]
Block Size	[32,1,1]
Registers/Thread	85
Shared Memory/Block	388 B
▾ Occupancy	
Theoretical	31.2%
▾ Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
▾ Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

Figure 5.27: The profiler for the kernel that casted 12.5% rays for each block

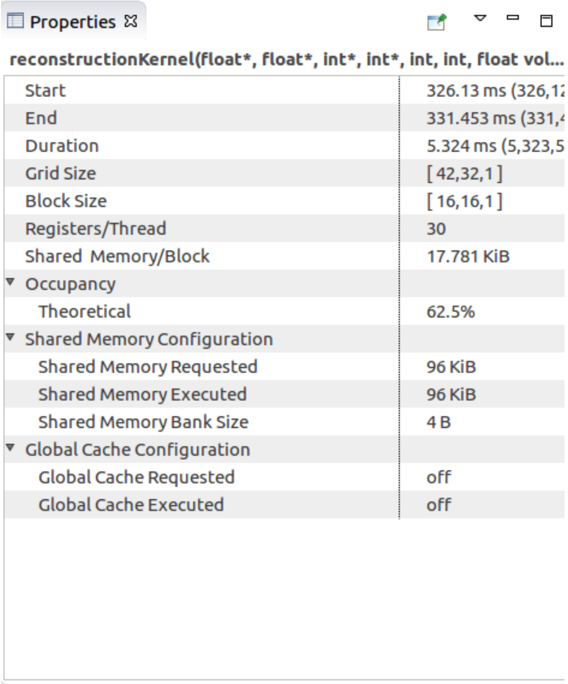


Properties

d_renderSecond(bool, float*, int*, float*, float*, float*, float*, ...

Start	1.122 s (1,122,413,
End	1.128 s (1,128,462,
Duration	6.049 ms (6,049,45
Grid Size	[42,32,1]
Block Size	[16,16,1]
Registers/Thread	83
Shared Memory/Block	0 B
▾ Occupancy	
Theoretical	25%
▾ Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
▾ Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

Figure 5.28: The profiler for the kernel that casted additional rays for each block



Properties

reconstructionKernel(float*, float*, int*, int*, int, int, float vol...

Start	326.13 ms (326,1
End	331.453 ms (331,4
Duration	5.324 ms (5,323,5
Grid Size	[42,32,1]
Block Size	[16,16,1]
Registers/Thread	30
Shared Memory/Block	17.781 KiB
▾ Occupancy	
Theoretical	62.5%
▾ Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
▾ Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

Figure 5.29: The profiler for the conjugate gradient solver kernel

From the above figures, it is obvious that the rays are cast in different ways but the profiler for the reconstruction kernel still has the same thread occupancy as it had for the local solver. The experimental results show that the GPU based compressive rendering methodology can save a significant amount of time for direct volume rendering while maintaining the high quality of images.

Chapter 6

Conclusion

6.1 Summary

In the world of high quality renderers, performance has always been lacking with a high level of interactivity. This thesis is intended to solve the question of achieving high quality with interactivity. The main focus of the thesis is to use an interactive conjugate solver to work with a high quality volume renderer. The use of compressive rendering lowers computational cost by discarding subsets of the pixels and within a few milliseconds a high quality image can be generated with a solver. The time that is saved from the volume renderer can significantly boost the overall performance of the compressive volume renderer. There are other criteria that can increase the performance of the volume renderer. One of these criteria is the number of iteration; a fixed number of iterations can produce a better performance but may affect the quality of the rendered image. On the other hand, if iterations are performed to a predefined condition it can result in a better quality image but will adversely affect the performance. Depending on the user requirements either techniques can be used to achieve quality or performance.

6.2 Contributions

In this thesis the focus is on the performance of the volume renderer and performance is augmented with GPU based compressive rendering. The thesis has shown that:

- A high quality image can be generated by only using a subset of pixels.
- Computational cost and time can be saved by discarding pixels and the pixels can be recovered using a conjugate gradient solver.

- Performance can be improved using three different techniques:

    - **Global Solver**: In this approach CUDA's library is used in FFT based convolution and vector dot products. The experimental results is significantly faster than the work done by Liu and Alim et al. [4] but this was not fast enough for real time interactive rendering.

    - **Local Solver**: In this method the data is partitioned into smaller blocks and using a new convolution strategy by accessing shared memory. Parallel reduction strategy is used for dot product operation and this produced a drastic improvement in performance.

    - **Adaptive Sampling**: In this strategy the amount of pixel distribution in each block is determined by variance analysis. This process also incorporates multiple passes; the first pass only casts rays for a minimum number of pixels for each block and then calculates the variance for each block. In the second pass, based on the variance analysis, a proportional number of pixels are added for ray casting. In the third pass, the gradient solver is used to reconstruct the entire image.

6.3 Future Work

The methodology in this thesis can be used in real time high resolution video streaming and the bandwidth of video streaming can be saved while streaming. The transmitting device and the receiving end will have the same binary sampling matrix and using this binary matrix the data will be transmitted. The receiving end can generate the full image by using the solver with the partial data received. This methodology can also be used on client server based rendering systems. At the server side an image with missing pixels will be generated and at the client side a conjugate gradient solver will recover all the missing pixels. Since this methodology can work with any size image there are instances where the data set may exceed a single GPU's memory (eg. 1TB of volume data) and may require a very high resolution display. This scenario can be overcome by using multiple GPUs; the GPUs can be physically arranged into a tree where the GPUs are connected to a PCIe switch. Then a fixed portion of the image can be allocated to a particular GPU; this GPU will divide the allocated image space into smaller blocks that will be the same size of the CUDA blocks.

Possible phases for multi-GPUs compressive rendering will be:

1. **Phase 1**: Allocate portion of screen space to each GPU.
2. **Phase 2**: Each GPU can share the same sampling matrix. Based on the sampling matrix, each GPU will generate a volume rendered image with unknown pixels.
3. **Phase 3**: Run the conjugate gradient solver to recover the missing pixels.

Since the local solver is independent to each CUDA block, it does not require any global synchronization. Therefore each individual CUDA block will write its' results to its' respective screen space.

Bibliography

[1] David Kirk and Wen-mei W. Hwu. *Programming massively parallel processors: a hands-on approach*. Elsevier, 2017.

[2] Michal Trybulec. `http://www.yac.com.pl/mt.graphics.ray-casting.en.html`.

[3] Cuda memory model. `http://geco.mines.edu/tesla/cuda_tutorial_mio/`. [Online; accessed 24th-August-2017].

[4] Xiaoyang Liu and Usman R Alim. Compressive volume rendering. In *Computer Graphics Forum*, volume 34, pages 101–110. Wiley Online Library, 2015.

[5] Wikipedia. Volume rendering. `https://en.wikipedia.org/wiki/Volume_rendering`. [Online; accessed 28th-August-2017].

[6] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.

[7] Marc Levoy. Display of surfaces from volume data: Ieee comput. graph. & appl. vol 8 no 3 (1988) pp 29–37. *Computer-Aided Design*, 20(7):425, 1988.

[8] Thomas Porter and Tom Duff. Compositing digital images. In *ACM Siggraph Computer Graphics*, volume 18, pages 253–259. ACM, 1984.

[9] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[10] The visualization toolkit. `https://www.vtk.org/`. [Online; accessed 20th-August-2017].

[11] Paraview. `https://www.paraview.org/`. [Online; accessed 20th-August-2017].

[12] visit homepage. `http://www.visitusers.org/`. [Online; accessed 20th-August-2017].

[13] Voreen Westflische Wilhelms-Universitt Mnster. Navigation:. `http://www.uni-muenster.de/Voreen/`, Feb 2017.

[14] Will J Schroeder, Bill Lorensen, and Ken Martin. *The visualization toolkit: an object-oriented approach to 3D graphics*. Kitware, 2004.

[15] Utkarsh Ayachit. *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc., USA, 2015.

[16] Parallel unstructured volume rendering in paraview. 2007.

[17] Harvey Ray, Hanspeter Pfister, Deborah Silver, and Todd A Cook. Ray casting architectures for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):210–223, 1999.

[18] Huamin Qu, Feng Qiu, Nan Zhang, Arie Kaufman, and Ming Wan. Ray tracing height fields. In *Computer Graphics International, 2003. Proceedings*, pages 202–207. IEEE, 2003.

[19] Timothy J Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3d texture hardware. 1993.

[20] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 Symposium on Volume Visualization*, VVS '94, pages 91–98, New York, NY, USA, 1994. ACM.

[21] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '00, pages 109–118, New York, NY, USA, 2000. ACM.

[22] William Donnelly. Per-pixel displacement mapping with distance functions. *GPU gems*, 2(22):3, 2005.

[23] J Dummer. Cone step mapping: An iterative ray-heightfield intersection algorithm (2006). *URL: http://www. lonesock. net/files/ConeStepMapping. pdf*, 2(3):4.

[24] Fabio Policarpo and Manuel M Oliveira. Relief mapping of non-height-field surface details. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 55–62. ACM, 2006.

[25] Jens Kruger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38. IEEE Computer Society, 2003.

[26] Daniel Cohen and Amit Shaked. Photo-realistic imaging of digital terrains. In *Computer Graphics Forum*, volume 12, pages 363–373. Wiley Online Library, 1993.

[27] Klaus Engel, Markus Hadwiger, Joe Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-time volume graphics*. CRC Press, 2006.

[28] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on visualization and computer graphics*, 8(3):270–285, 2002.

[29] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '01, pages 9–16, New York, NY, USA, 2001. ACM.

[30] E Swan and Roni Yagel. Slice-based volume rendering. *OSU-ACCAD-I/93-TR1,*, 1993.

[31] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of the Symposium on Data Visualisation 2003*, VISSYM '03, pages 231–238, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[32] NVIDIA. Volume rendering with 3d textures. `http://docs.nvidia.com/cuda/cuda-samples/index.html#volume-rendering-with-3d-textures`.

[33] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Straßer. Interactive rendering of large volume data sets. In *Visualization, 2002. VIS 2002. IEEE*, pages 53–60. IEEE, 2002.

[34] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177. ACM, 1998.

[35] Han-Wei Shen, Ling-Jen Chiang, and Kwan-Liu Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In *Proceedings of the conference on Visualization'99: celebrating ten years*, pages 371–377. IEEE Computer Society Press, 1999.

[36] Eric B Lum, Kwan Liu Ma, and John Clyne. Texture hardware assisted rendering of time-varying volume data. In *Proceedings of the conference on Visualization'01*, pages 263–270. IEEE Computer Society, 2001.

[37] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16. ACM, 2001.

[38] Jens Schneider and Rudiger Westermann. Compression domain volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 39. IEEE Computer Society, 2003.

[39] Enrico Gobbetti, José Antonio Iglesias Guitián, and Fabio Marton. Covra: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks. In *Computer Graphics Forum*, volume 31, pages 1315–1324. Wiley Online Library, 2012.

[40] P. Sen and S. Darabi. Compressive rendering: A rendering application of compressed sensing. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):487–499, April 2011.

[41] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[42] Michael Garland and David B Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010.

[43] Wikipedia. Parallel programming model. `https://en.wikipedia.org/wiki/Parallel_programming_model`. [Online; accessed 5th-August-2017].

[44] Dr. John Owens. Cuda home page. `http://www.nvidia.com/object/cuda_home_new.html`.

[45] Cuda programming blogspot. `http://cuda-programming.blogspot.ca/2013/02/texture-memory-in-cuda-what-is-texture.html`. [Online; accessed 24th-December-2017].

[46] Peter G Anderson. Linear pixel shuffling for image processing: an introduction. *J. Electronic Imaging*, 2(2):147–154, 1993.

[47] Xie Xu, Alexander Singh Alvarado, and Alireza Entezari. Reconstruction of irregularly-sampled volumetric data in efficient box spline spaces. *IEEE transactions on medical imaging*, 31(7):1472–1480, 2012.

[48] Thierry Blu, P Thcvenaz, and Michael Unser. Moms: Maximal-order interpolation of minimal support. *IEEE Transactions on Image Processing*, 10(7):1069–1080, 2001.

[49] Magnus Rudolph Hestenes and Eduard Stiefel. *Methods of conjugate gradients for solving linear systems*, volume 49. NBS, 1952.

[50] Victor Podlozhnyuk. Fft-based 2d convolution. *NVIDIA white paper*, page 32, 2007.

[51] Harris Mark. Optimizing parallel reduction in cuda. *NVIDIA CUDA SDK*, 2, 2008.