Design and Implementation of Video View Synthesis for the Cloud

by

Parvaneh Pouladzadeh

Thesis submitted

In partial fulfillment of the requirements

For the Master of Applied Science in Electrical and Computer Engineering

School of Electrical Engineering and Computer Science

Faculty of Engineering

University of Ottawa

Abstract

In multi-view video applications, view synthesis is a computationally intensive task that needs to be done correctly and efficiently in order to deliver a seamless user experience. In order to provide fast and efficient view synthesis, in this thesis, we present a cloud-based implementation that will be especially beneficial to mobile users whose devices may not be powerful enough for high quality view synthesis. Our proposed implementation balances the view synthesis algorithm's components across multiple threads and utilizes the computational capacity of modern CPUs for faster and higher quality view synthesis. For arbitrary view generation, we utilize the depth map of the scene from the cameras' viewpoint and estimate the depth information conceived from the virtual camera. The estimated depth is then used in a backward direction to warp the cameras' image onto the virtual view. Finally, we use a depth-aided inpainting strategy for the rendering step to reduce the effect of disocclusion regions (holes) and to paint the missing pixels. For our cloud implementation, we employed an automatic scaling feature to offer elasticity in order to adapt the service load according to the fluctuating user demands. Our performance results using 4 multi-view videos over 2 different scenarios show that our proposed system achieves average improvement of 3x speedup, 87% efficiency, and 90% CPU utilization for the parallelizable parts of the algorithm.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Professor Shervin Shirmohammadi for his outstanding guidance, support and motivation since I joined this program and throughout the writing of this dissertation. His meticulous supervision has not only educated me in research work but it will also serve as an inspiration and a model to follow in the future. Without his guidance and support, this dissertation would not have been possible.

Many thanks go to my co-supervisor Dr. Razib Iqbal, who spent a considerable amount of his time with me in brainstorming ideas, approaches and valuable discussions throughout this research. I would also like to thank Dr. Omid Fatemi for his continuous support, motivation and valuable discussions through my graduate studies.

Last but not the least, I would like to thank my friends and my family for the spiritual support, constant encouragement and unconditional love they have been providing throughout my life.

Dedication

This Thesis is dedicated to my parents.

For their endless love, support, and encouragement.

Table of Contents

List of Figures

List of Tables

List of Acronyms

2D	Two Dimentional Space
3D	Three Dimentional Space
3DTV	Three Dimentional Television
API	Application Program Interface
ASG	Auto Scaling Group
AWS	Amazon Web Service
CCD	Charged Coupled Device
CPU	Central processing unit
EC2	Elastic Compute Cloud
ELB	Elastic Load Balancing
FTV	Free Viewpoint Television
FVV	Free Viewpoint Video
GPU	Graphics processing unit
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a service
IBR	Image Based Rendering
ILP	Instruction Level Parallelism
IT	Information Technology
MPDP	Multi pass Dynamic Programming
NIST	National Institute of Standards and Technology

PaaS	Platform as a Service
PC	Personal Computer
POSIX	Portable Operating System Interface
QoS	Quality of Service
RGB	Red Green Blue
SMV	Super Multi-View
SaaS	Software as a Service
SLA	Service Level Agreement
SMV	Super Multi view
SOA	Service Oriented Architecture
SPI	SaaS PaaS IaaS
OpenCV	Open Source Computer Vision

Chapter 1

Introduction

Computer vision allows computing systems to better perceive and comprehend their environment. In recent years, Free Viewpoint Television (FTV) and Three Dimensional Television (3DTV) have become subjects of both academic research and industry interest. Among these applications, the question of how to understand the 3D information from a scene, obtained from multiple cameras, has attracted the attention of many researchers. Systems for rendering arbitrary views of natural scene have been well known in computer vision community for a long time, but in very recently years the speed and quality reached a desired level of end user interest.

One prominent example of the impact of these recent advances in computer vision, is emerging advances of 3D displays called Super Multi-View (SMV), which render hundreds of horizontal parallax ultra-dense views, providing high quality, glasses-free, and 3D viewing experience with a wide viewing angle and without eye fatigue, smooth transition between adjacent views, and even some walk-around feeling on foreground objects [1][2][3].

For better performance, FTV is a visual media that allows users to view a 3D scene by freely changing the viewpoint. Figure 1.1 shows the multi-camera positions which can be set anywhere and displayed views are not camera views but generated views. Free viewpoint images have been generated in the past by a Personal Computer cluster in [4]. Now, they can be generated in real time by a single PC [5] or a mobile player, as shown in Figure 1.2, which exemplifies the technical advances made possible by the rapid progress

of the computational power of processors. Figure 1.2 shows FTV on laptop PC with mouse control, FTV on a mobile player with touch panel control, FTV on an all-round 3D display(seelinder), FTV on a 2D display with head tracking, FTV on a multi-view 3D display without head tracking and FTV on a multi-view 3D display with head tracking.

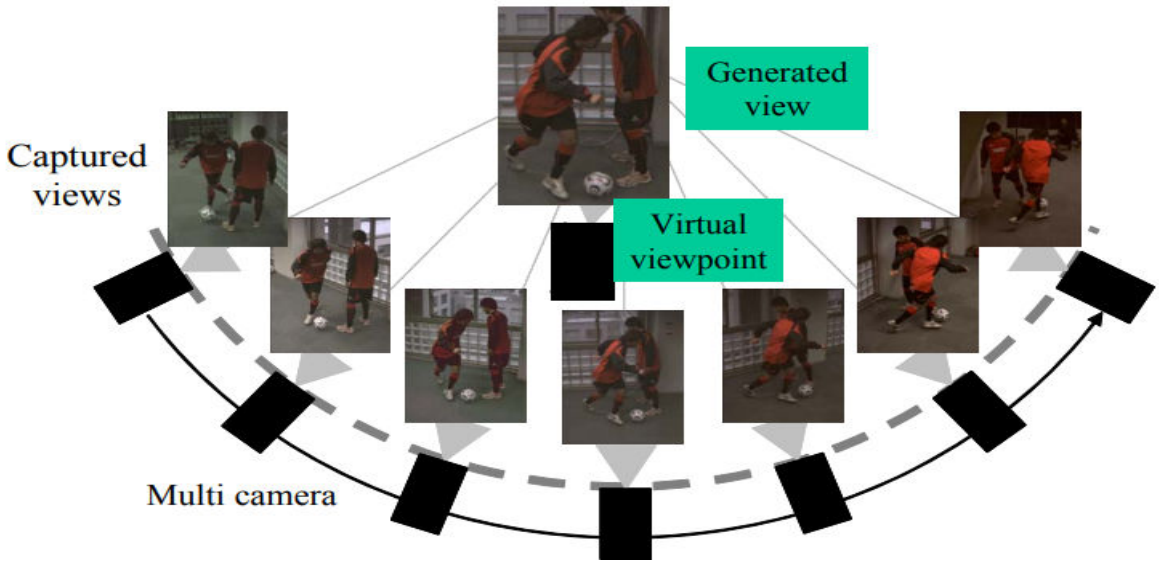

Figure 1.1: Example of FTV showing generated views by multi-camera [1]

Figure 1.2: Example of FTV on different platforms and user interfaces [1]

It is noteworthy that Free Viewpoint Video (FVV) / Free Viewpoint Television (FTV) on mobile devices over cellular networks is very challenging due to the requirement for large bandwidth and limitations in computation and battery life on mobile phones [6]. As such, we propose to outsource the view synthesis task at the server side, and then simply stream the virtual view to the mobile user. In fact this is the trend we are seeing, for example in third generation 3D/multi-view standards, where the view synthesis task, which was at the users side in the first and second generation, are now moved to the server side.

With FTV, viewers can choose the angle and position from which to view a scene. For instance, free navigation videos with accompanying spatial audio, as shown in Figure 1.3, could be delivered through the Internet as a new service, which may even use low-power mobile devices.

Figure 1.3: Example of FTV mobile application [1]

By emerging cloud computing, multimedia cloud computing can provide a variety of computing and storage services for mobile devices [7]. To this end, to serve a large number of clients, a seamless framework is needed to meet the growing demand for view synthesis applications. Cloud-based platforms offer an on-demand solution for these applications. By running the view synthesis module on a powerful server, we can generate a higher quality virtual view of higher frame rate. However, to truly make the system scalable, fast,

and efficient, a server form or a cloud environment should be used rather than a single server system. This enables parallelizing and distributing the view synthesis module.

1.1 Motivation

Television realized the human dream of seeing a distant world in real time and has been served as the most important visual media to date [1]. TV provides us the same scene even if we change our viewpoint in front of the display. This function of TV has been appeared by view synthesis or image-based rendering (IBR) technique that synthesizes realistic, novel views directly from input views without the full 3D scene model [8]. This is quite different from what we experience in the real world. Users with TV can get only a single view of a 3D world. So, the view is determined not by users but by a camera placed in the 3D world [1].

To create such a high fidelity user experience, a very dense and a wide baseline set of views needs to be captured, encoded, transmitted and displayed. This puts very demanding requirements on storage, codec, and bandwidth, in order to handle hundreds of wide baseline, ultra-dense views [3].

Additionally, since these applications allow the user to view the video from any arbitrary angle, the system must be able to do the view synthesis, i.e. recover arbitrary views (virtual views) not physically captured with the camera. Considering that multi-view geometry processing required for view synthesis is a computationally expensive task, especially given the high resolution of modern FTV systems, it puts the view synthesis ability out of reach of most users who today access their entertainment on mobile devices that do not have the required graphics computing and processing power.

1.2 Problem Statement

In this work, we study the feasibility of outsourcing view synthesis applications to the cloud environment and focus on different strategies for load balancing through cloudlets.

Cloudlets is a term referring to mobility-enhanced small-scale cloud resources. We analyze parallel implementations of prominent components of the algorithm as well as its data dependencies, data race conditions and synchronization. Different levels of parallelism are also taken into account such as task level, data level and instruction level parallelism. Once the system is deployed on the cloud, we will demonstrate, through a complete process of experimentation, the effectiveness of each scenario by evaluating the quantitative metrics of a parallel program, including speedup, efficiency and utilization factor. Figures 1.4 and 1.5 show the examples of how our reference view synthesis software [2], as the baseline generates novel viewpoints of input images and depth maps, as well as the virtual view output.

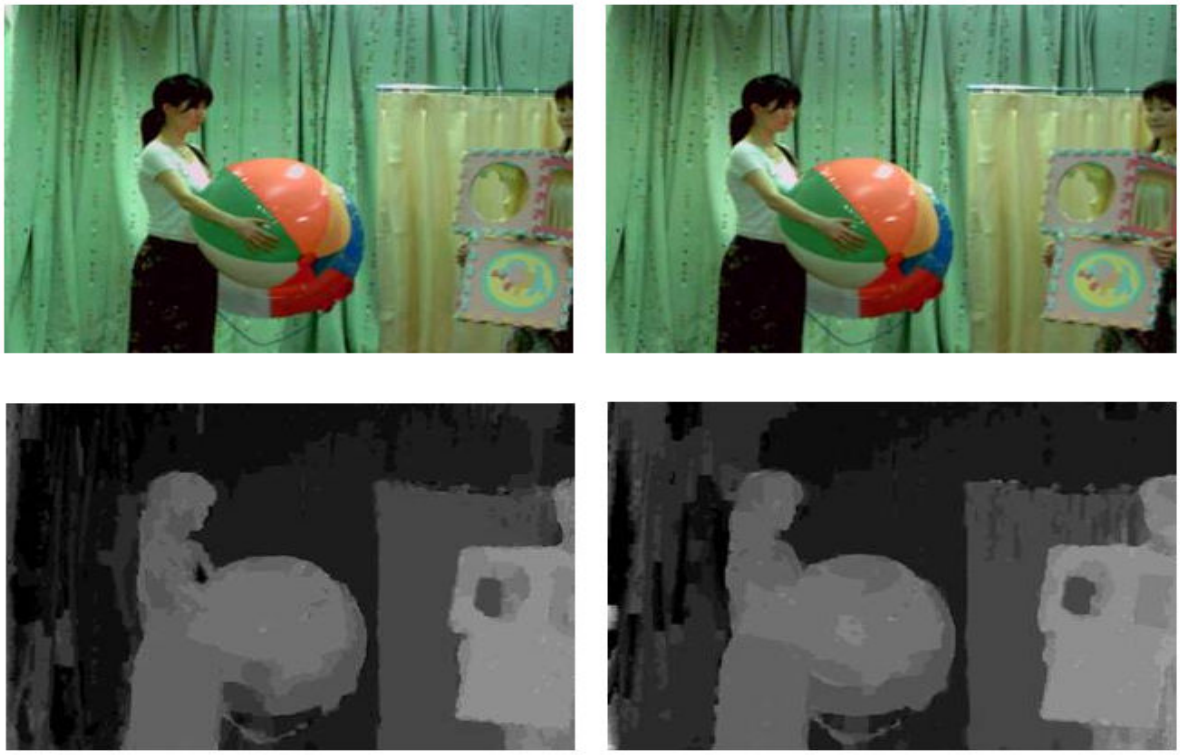

Figure 1.4: Example of multi-view depth images from the video sequence [2]

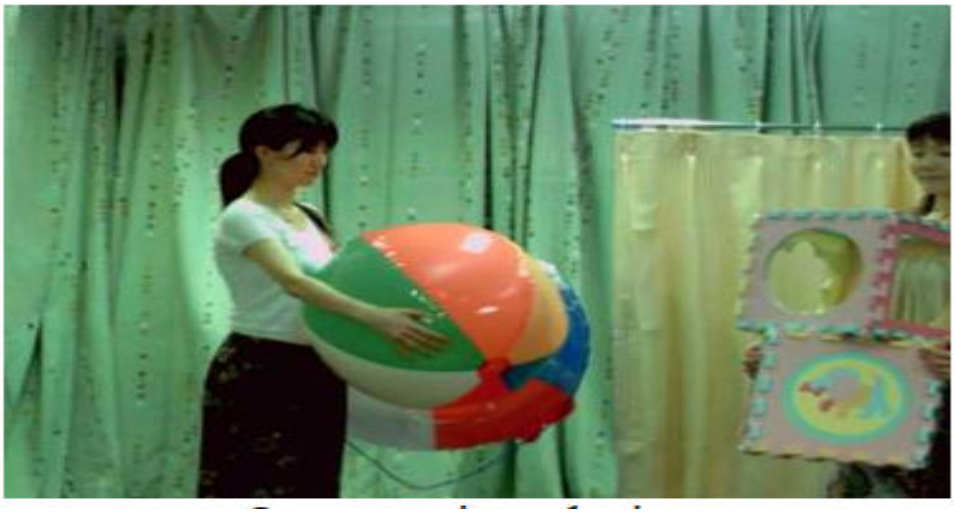

Figure 1.5: Example of output virtual view [2]

Our view synthesis algorithm [2] accepts images of real world scene captured from multiple cameras along with corresponding depth information to generate a virtual image from a novel viewpoint. This approach is based on homography computing and provided depth maps. The depth maps of the 3D scene can be either computed using computer vision approaches or recorded by RGB sensors. The main contribution of the present research is propose a view synthesis algorithm that can be parallelized over multiple processors. To this end, an efficient parallel implementation that only requires modern CPUs (not expensive GPUs). We also propose deploying the view synthesis system into cloud platforms and we have investigated the realization of prominent parts of the algorithm while preserving reasonable load balance across processing resources. Due to the fact that there are several possible scenarios for multi-threaded implementation, each of them needs to be evaluated by quantitative metrics for a parallel program. Therefore, we have evaluated possible multi-threaded implementation scenarios using speedup, efficiency and utilization reported with respect to one another.

1.3 Contributions

The main research contributions of this thesis are summarized as follows:

- We designed a view synthesis system suitable for parallel processing on cloud platforms. Our system based on homography estimation accepts images captured from multiple cameras along with their corresponding depth information to generate a virtual image from an arbitrary viewpoint.

- We also implemented and tested our view synthesis method on the cloud with different strategies for parallelization and synchronization. We comprehensively validated the effectiveness of proposed implementation over the sequential version.

- For an on demand user experience, we added an auto-scaler and load balancer capabilities to our cloud implementation in order to automatically adapt server resources based on users demand. We have validated our design by constant monitoring of server loads and metrics.

- We then evaluated the results based on quantitative metrics such as speedup, efficiency and resource utilization. Experimental results show that our proposed system achieves 3x speedup, 87% efficiency, and 90% CPU utilization for the parallelizable parts of the algorithm.

1.4 List of Publications

During the process of completing this work, the following conference paper has been published:

- Parvaneh Pouladzadeh, Razib Iqbal, Shervin Shirmohammadi, Omid Fatemi, ***A Cloud-based Multi-Threaded Implementation of View Synthesis System***. The 19th IEEE International Symposium on Multimedia (ISM), Taichung, Taiwan, 2017.

1.5 Thesis Organization

From the following chapter onwards, we explore crucial steps toward an elastic multi-threaded view synthesis system. Our objective in this thesis aims at feasibility of bringing view synthesis application on the cloud platform and evaluation of various parallel implementation scenarios based on quantitative metrics.

- In Chapter 2 , background information of the system which is fundamental for understanding the work presenting in this thesis are categorized and explained. It starts with the presenting of view synthesis components, multi-threading and cloud computing in general.
- In Chapter 3 , we survey previous research in view synthesis and elasticity in cloud computing as a required background needed for readers, who are not familiar with the concept.
- In Chapter 4 , we present our multi-threaded view synthesis implementation based on two major parallel scenarios. We also analyzed the elasticity of our deployed application and assessed its reliability using AWS CloudWatch service.
- In Chapter 5 , we summarize our experimental results supporting the multi-threaded view synthesis implementation and analyze its elasticity on the cloud.
- In Chapter 6 , we conclude the thesis and state the main findings of our research. Also potential works for future research are summarized.

Chapter 2

Background

2.1 Introduction

We will start this chapter by classifying the view synthesis components including background information regarding the pinhole camera model, the perspective projection, intrinsic/extrinsic parameters, warping, depth map, homography and hole-filling characteristics. This will allow us to understand the necessary components for generating a virtual view. Afterwards, the background information regarding multi-threading will be presented. Lastly, in this chapter we will describe some critical information regarding cloud computing in general, with a focus on the Amazon Web Services (AWS) cloud environment and its services, including Elastic Compute Cloud (EC2), Auto-scaling Group (ASG), Elastic Load Balancing (ELB) and CloudWatch.

2.2 View Synthesis Overview

A view synthesis system is based on a model that consists of a set of images of a scene with their corresponding depth maps. When the depth of each point in an image is known, the image can be rendered from any nearby arbitrary point of view by projecting the pixels of the image to their proper 3D locations and projecting back onto a new image plane. Thus,

a new set of images is generated by warping each image according to its depth map [9]. To this end, the challenge of recovering arbitrary views of natural scenes through multi-camera systems has received considerable attention from the computer-vision community. If this challenge is met, it would introduce a new dimension to how humans experience the surrounding world. Synthesizing a novel view requires at least two views of a scene, as well as a method for obtaining 3D geometry through the incorporated cameras. The essential components of a view synthesis scheme are elaborated in the following subsections. The following Figure 2.1 schematically shows the essential components of our view synthesis system.

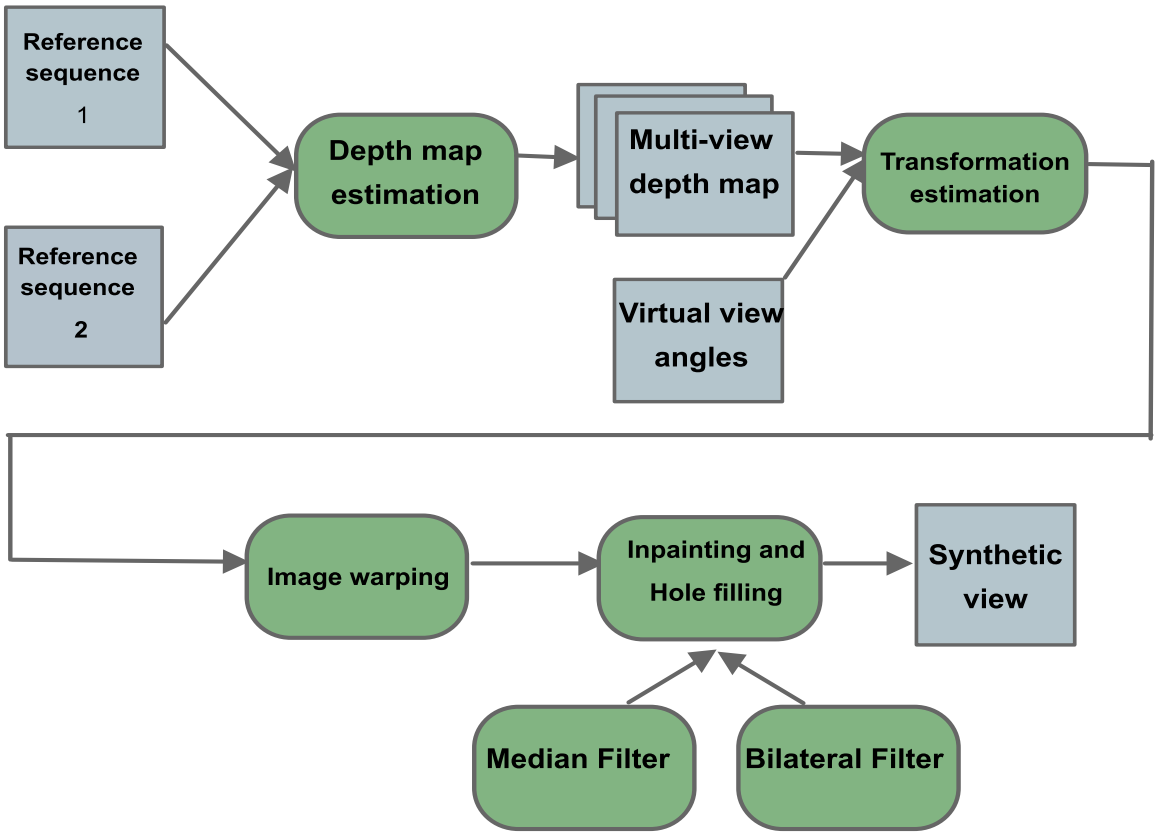

Figure 2.1: View synthesis block diagram

2.2.1 Pinhole camera model

The main purpose of the pinhole camera model is to transform the 3D image coordinates to 2D image coordinates. This transform is determined by the cameras extrinsic and intrinsic parameters. 2D image correlation methods are used to transform from 3D to 2D images. Accurate camera-model parameters are used for each camera in transformation [10].

The position and orientation of a camera are represented by a translation matrix and rotation matrix $[R|t]$, respectively. To parameterize these matrices an in-depth study of a pinhole cameras intrinsic and extrinsic parameters is needed. There are various multi-sensory approaches [11][12] exist, e.g., algebraic methods that are based on a single visual sensor, such (CCD camera). These methods require a set of 2D points in the image plane and the coordinates of the corresponding 3D scene points.

Figure 2.2, illustrates perspective transformation from homogeneous 3D points to a 2D homogeneous coordinates.

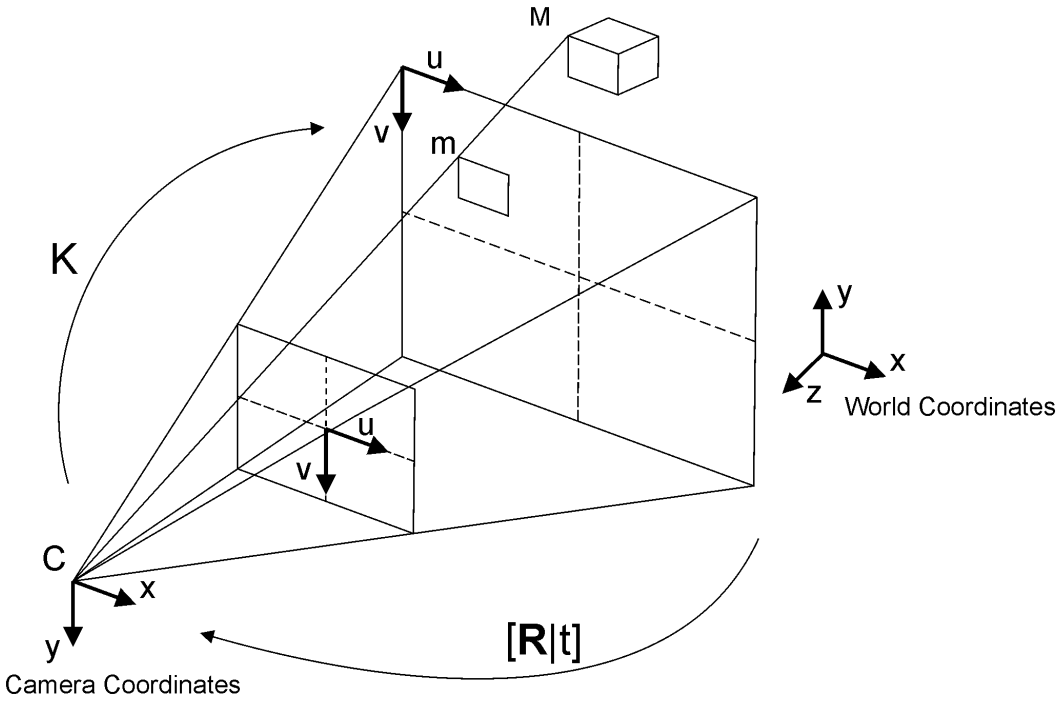

Figure 2.2: Perspective view of pinhole camera model.

2.2.2 The perspective projection

A realistic model for standard pinhole camera model can be formulated as a perspective projection. To be more specific, image pixels are first projected into a set of 3D world points, and then these points in the world coordinates are projected back to the 2D image plane of the virtual camera. This can be expressed by a $\mathbb{R}^3 \rightarrow \mathbb{R}^2$ mapping from a 3D point $M = [X, Y, Z, 1]^T$ in homogeneous coordinates to a 2D pixel $m = [u, v, 1]^T$ in image plane as shown in (2.1).

$$m = PM, \tag{2.1}$$

where $P_{3\times4}$ is called projection matrix that parametrizes cameras' relative translation (t) and orientation (R) such that (2.1) becomes (2.2).

$$m = K[R|t]M, \tag{2.2}$$

where $K_{3\times3}$ denotes the camera matrix. Figure 2.2 demonstrates the graphical representation of this projection.

2.2.3 Intrinsic parameters

By referring to Figure 2.3, the intrinsic parameters, also known as camera matrix, is defined as a 3-by-3 matrix to map 3D camera coordinates to the 2D image pixels. Using this mapping technique the axial focal length that is represented in pixels can be formulated. The process of finding these parameters is usually done offline and called camera calibration. The camera intrinsic parameter denoted by K in the form of (2.3),

$$k = \begin{bmatrix} k_u f & 0 & c_u \\ 0 & k_v f & c_v \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.3}$$

where $k_u f$ and $k_v f$ are horizontal and vertical focal length respectively. k_u and k_v are two scale factors expressing pixel density of the camera's CCD sensor. c_u and c_v are camera principal points ideally indicating the center of image plane. As mentioned earlier, extrinsic parameters are used to formulate camera's position (t) and orientation (R) which can translate 3D world coordinates to 3D camera coordinates [13].

2.2.4 Extrinsic parameters

Extrinsic parameters explain the camera position and orientation in world coordinates. These parameters can be expressed as 3×4 matrix $[R|t]$ comprising of a rotation matrix $R_{3\times3}$ and translation matrix $t_{3\times1}$ [13]:

$$[R|t] = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \tag{2.4}$$

the transformation from a 3D point in the camera coordinates (M_w) to the world coordinates (M_c) is expressed as:

$$M_w = R^{-1}(M_c - t) = R^{-1}M_c - R^{-1}t \tag{2.5}$$

where, $-R^{-1}t$ is the position of camera center in world coordinates. We can write as:

$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \begin{pmatrix} X_w + t_1 \\ Y_w + t_2 \\ Z_w + t_3 \end{pmatrix} \tag{2.6}$$

2.2.5 Distortion Coefficients

Determining the intrinsic and extrinsic parameters of the camera would usually suffice for estimating the camera pose, in some cases, we cannot neglect the distortion caused by camera lens. This distortion can be compensated by 2D displacement terms called radial distortion and decentring distortion. Let $\mathbf{x}_u$ and $\mathbf{x}_d$ be un-distorted and distorted image point respectively [13]. We can write:

$$x_u = x_d - d_{radial} - d_{decenter} \tag{2.7}$$

Although the distortion caused by decentring is usually ignored, the radial distortion can be determined by a polynomial expression as follows:

$$d_{radial} = 1 + k_1 r^2 + k_2 r^4 + \dots \tag{2.8}$$

Where r is the radial distance from the center of image plane.

2.3 View Synthesis (Warping)

Image warping is the process of transforming images as if they are viewed from a different viewpoint. Camera pose matrix is treated as the transformation matrix to warp images. For more realistic transformation, camera lens distortion is sometimes taken into account as well as intrinsic and extrinsic parameters. In fact, image warping specifies where each pixel goes in the destination image by using the transformation matrix. To discretize the pixel locations in the warped images, different interpolation strategies such as bilinear interpolation is employed. When practically implementing image warping, as treated as image pixels as a vector of 2D homogeneous points. So the intensity and location of each points can be determined using homography estimation. As shown in Figure 2.3, 2D pixels seen from each camera need to be described in the image plane of a virtual camera. This can be done using the projection matrix P and its inverse P^{-1}. Therefore, we can write

(2.9).

$$m'_v = P_v P_l^{-1} m'_l = P_v P_r^{-1} m'_r, \tag{2.9}$$

where P_v, P_l and P_r are projection matrices for virtual, left and right views, respectively. m'_v, m'_l and m'_r represents the non-homogeneous coordinates.

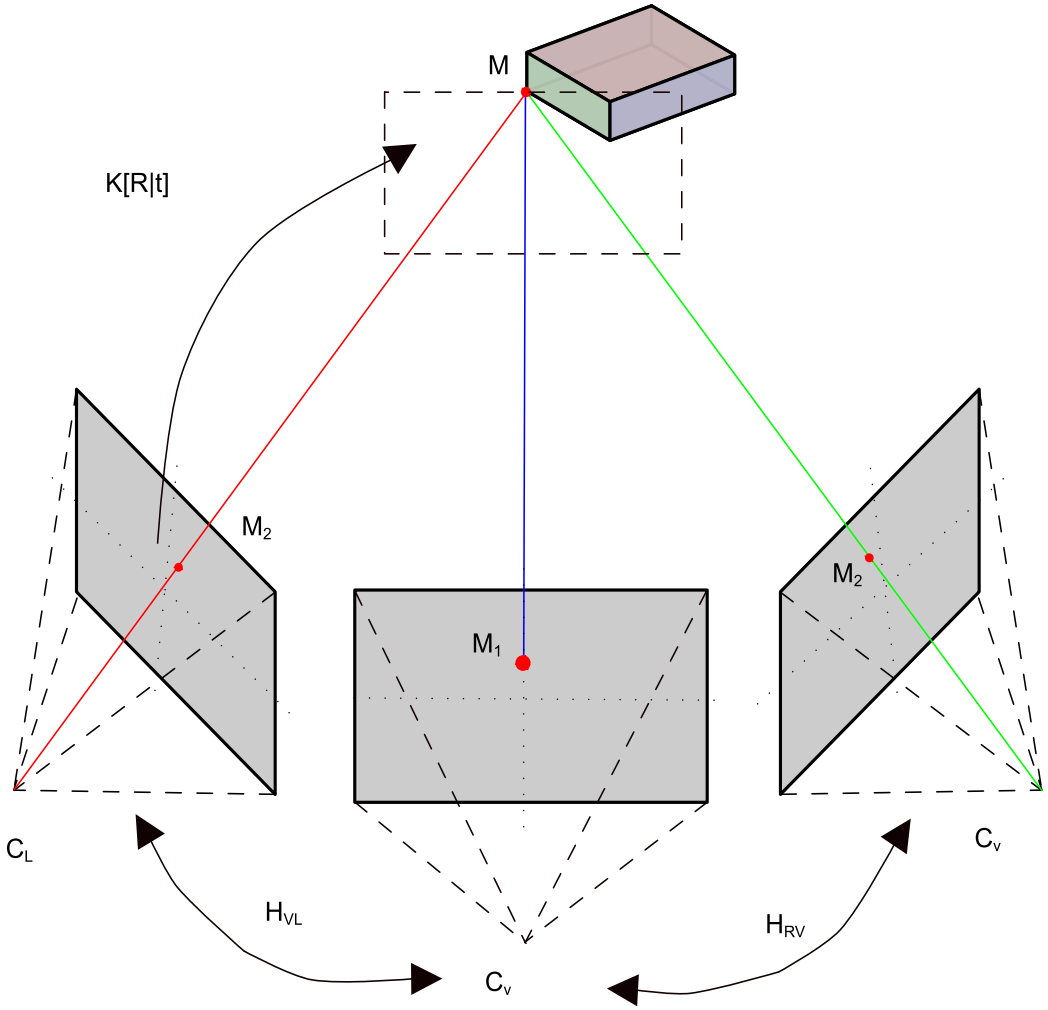

Figure 2.3: 2D- to-3D and 3D-to-2D transformation

2.3.1 Depth estimation

Depth information is needed to represent the distance of a coordinate in the 3D plane from the camera that it is seen through. Usually, an 8-bit channel is sufficient to store this

information, and the image representing this data is so-called depth map [14]. There are multiple ways to obtain this information using pairwise image matching including feature based matching, stereo matching, and block matching. In this work, we assume that this information is available to us.

2.3.2 Homography Transformation

In a special case where points are related by a plane-to-plane mapping, this transformation in a projective space is algebraically defined by a non-singular homography matrix $H_{3\times3}$ that maps two views of a planar object. Therefore, depth information of 3D points and their corresponding homographies are incorporated, and can be rewritten as in (2.10).

$$z_v m_v = H_v m_l^{-1} m_l z_l = H_v H_r^{-1} m_r z_r, \tag{2.10}$$

where z_v, z_l and z_r are depth values and m_v, m_l and m_r are defined in homogeneous coordinates.

2.3.3 Warping and Inpainting

Once homography relations are estimated for various depths in the scene, original images are warped to generate the virtual view in between. Novel view generation is achieved by estimating depth information of the scene from the virtual viewpoint, followed by median and bilateral filtering steps to smooth out discontinuities in the generated depth map. The new depth information is then used in a backward direction to warp reference frames (a left frame and a right frame) onto the virtual view. Depending on the relative position of the virtual camera, some level of disocclusion may be introduced, and other rendering artifacts may appear in the generated view. There are many techniques, e.g. [15] and [16] for dealing with hole-filling related to this disocclusion. However, we used depth-aided inpainting [2] in our implementation, which exploits the context around disocclusions that are visible from reference images to predict and in-paint missing pixels. Therefore, the

destination depth is the result of a weighted average over estimated values from the left and right frames; the destination depth is used as a guide to prioritize inpainting colors.

2.3.4 Hole-filling

In 3D warping, holes are often observed in the virtual view; thus, after generating an image from the merging process, the holes that have appeared need to be filled. Hole-filling operations give us the final form of the synthesized virtual view. Since a synthesized view from multiple reference images cannot comprehensively determine a 3D scene, a cloud of points is utilized. In addition, every location without sufficient color information is treated as a latent pixel and estimated accordingly. In this way, holes in a primary synthetic view are filled in the final output. There are two main kinds of holes. The first kind is a disocclusion hole, for which the corresponding pixel lies in the reference view. It is occluded by a pixel of another object closer to the camera. Disocclusion holes can be filled using depth-based image inpainting techniques [17][18]. The second kind, an expansion hole, is a spatial area of an objects surface in the virtual view whose corresponding area in the reference view is visible but has a smaller size. Expansion holes, unlike disocclusion holes, can leverage the information from neighboring pixels with similar depth for the purpose of interpolation [19].

2.4 Multi-threading (Computer Architecture)

According to [20], multi-threading is one of the core features supported by Java. It allows the creation of multiple objects that can simultaneously execute different operations. It is used in writing programs that perform different tasks, such as printing and editing, asynchronously. For instance, network applications that are responsible for serving remote clients requests have been implemented as multi-threaded applications. Computers are able to execute multiple programs at the same time owing to multiple cores and multi-threading functionality. Multi-threading is an amazing method to increase a programs performance; however, it is challenging to execute properly. Each thread is an independent set of values

for the processors registers that controls what executes and in what order, within the same program and having access to the same performance and memory resources. The use of multi-core processors in modern computers means that separate threads can be executed by separate cores or CPUs simultaneously [21].

Parallel computing has been traditionally associated with the HPC (high performance computing) community, but it is becoming more prevalent in mainstream computing due to the recent development of the commodity multi-core architecture. Consequently, it is expected that future generations of applications will heavily exploit the parallelism offered by the multi-core architecture.

There are two major approaches to parallelize applications, namely, auto-parallelization and parallel programming; the auto-parallelization approach, e.g. instruction level parallelism (ILP) or parallel compilers, automatically parallelizes applications that have been developed using sequential programming models. The advantage of this approach is that existing/legacy applications need not be modified. Therefore, programmers need not learn new programming paradigms. However, this is also becoming a limiting factor in exploiting a higher degree of parallelism, since it is extremely challenging to automatically transform algorithms of a sequential nature into parallel ones [21].

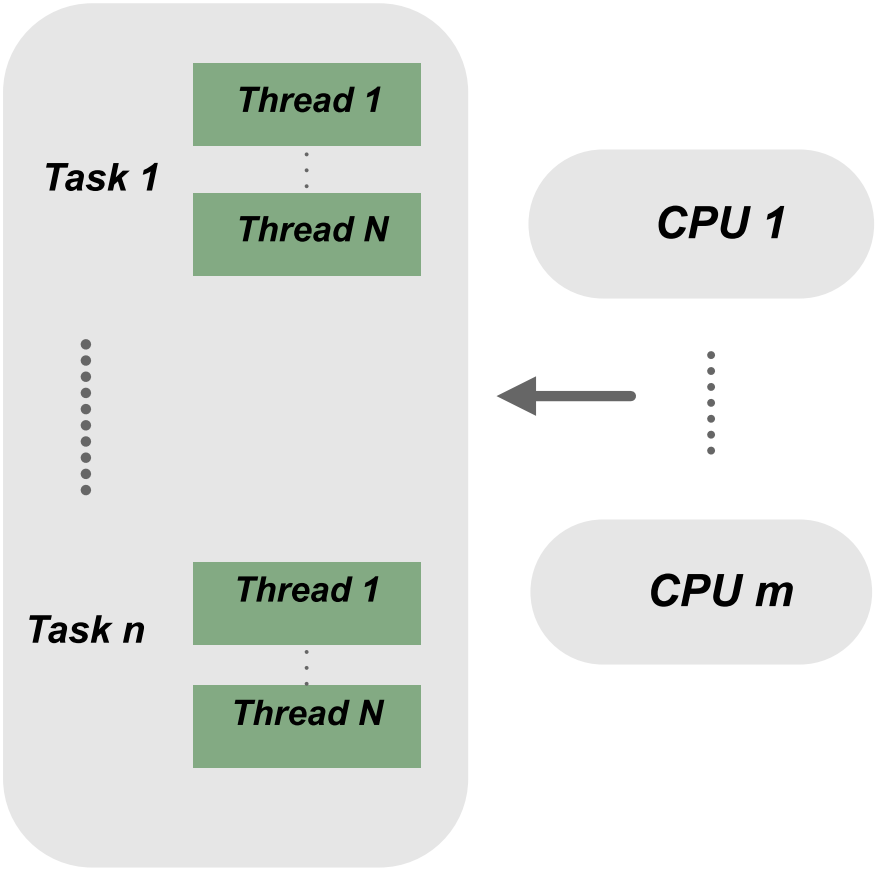

Figure 2.4: Multi-threading architecture

2.4.1 Synchronization

There are many applications where multiple threads might need to share a resource, possibly leading to unexpected results due to concurrency issues; hence, it is necessary to synchronize these threads in order to avoid critical resource confliction [22]. Otherwise, conflicts may arise when parallel threads try to modify a common variable at the same time [23]. For instance, when one thread starts executing the critical section, the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, a race condition may occur, in which the values of variables may be unpredictable and vary depending on the timings of the threads or of context switches within the processes [22].

A race condition, occurs when multi-threading is done using a distributed environment

or when there are interdependencies in the shared resources. Race conditions often lead to bugs, as these events happen in a manner that the system or programmer never intended. It can often result in a device crash, error notification, or shutdown of the application [24]. In other words, race conditions happen when two threads access a shared variable at the same time [25].

One example of activity synchronization methods is barrier synchronization shown in Figure 2.5. In this method, a complex computation can be divided and distributed among tasks. To this end, one task can finish its partial computation before other tasks complete theirs, but this task must wait for all other tasks to complete their computations before the task can continue.

As shown in Figure 2.5, a group of five tasks participate in barrier synchronization. Tasks complete their partial execution and reach the barrier at different times; however, each task in the group must wait at the barrier until all other tasks have reached the barrier threads [26].

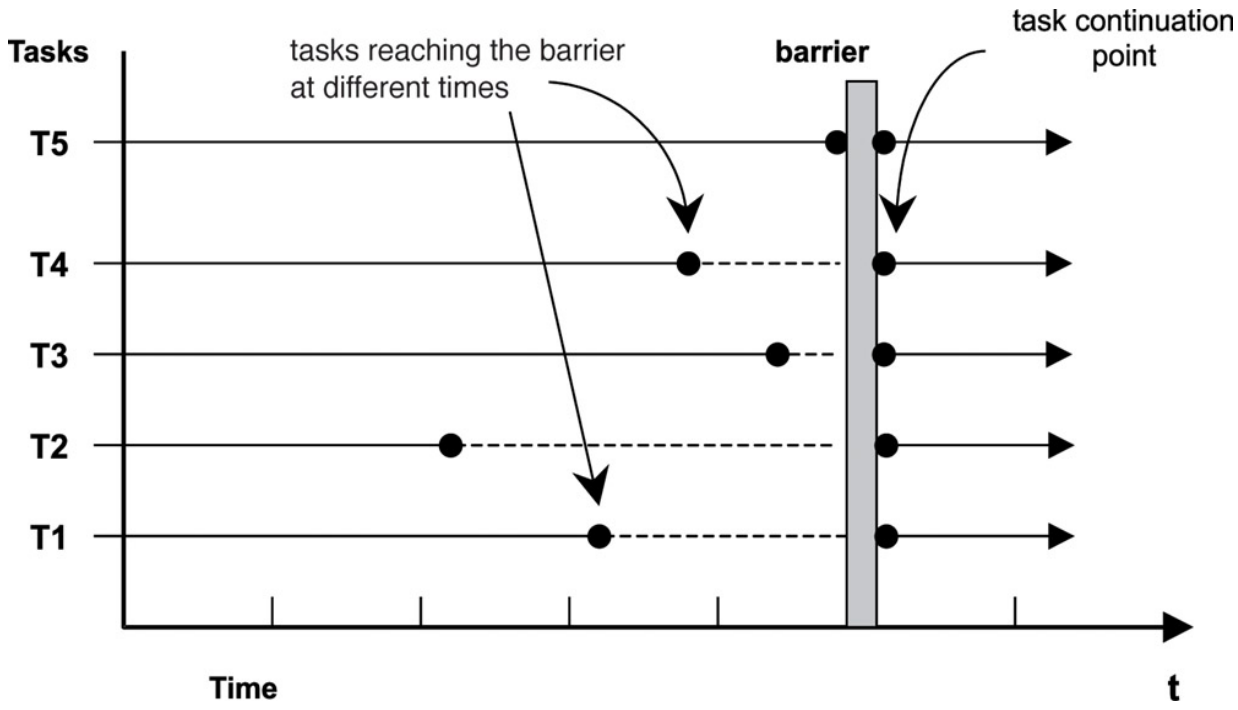

Figure 2.5: Visualization of barrier synchronization [26]

2.4.2 Pthread Library

In shared-memory multi-processor architectures, threads can be used to implement parallelism. POSIX Threads, usually referred to as pthreads, is an acronym for Portable Operating System Interface Threads and is a family of standards developed by the Institute of Electrical and Electronic Engineers (IEEE). Conceptually, POSIX describes a set of fundamental services needed for the efficient construction of application programs. Access to these services has been provided by defining an interface, using the C programming language, a command interpreter, and common utility programs that establish standard semantics and syntax. Pthreads are specified as a types of C language programming and procedure calls, implemented with a "pthread.h" header file. Worker management in pthreads requires the programmer to explicitly create and destroy threads by making use of pthreads' "create" and "exit" functions [20][21].

2.5 Cloud Computing Overview

There are several definitions of cloud computing, of which one concise definition is by US National Institute of Standards and Technology (NIST) [27].

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

According to Vouk [28], cloud computing is the combination of virtualization, distributed computing and the service-oriented architecture creates a new computing paradigm. Cloud computing is a growing idea in the world of IT, born out of the necessity for computing [29]. It is a popular trend in todays technology and refers to the storing and accessing of data over the Internet rather than computers hard drive [26]. It brings the user access to data, applications and storage that are not stored on their computer [29]. In addition, it helps clients build sophisticated and scalable applications on such resources. This

means clients do not have access the data from either their computers hard drive or over a dedicated computer network. In other words, cloud computing means data is stored at a remote place and is synchronized with other web information. For instance, Amazon, Google, Yahoo, and others have built large architectures to support their applications and, in turn, have taught the rest of the world how to build massively scalable architectures to support computing, storage, and application services. Vaquero [30] defines three major scenarios in cloud computing, which briefly introduced several cloud service models [31].

2.5.1 Cloud Service Models

Cloud computing could be classified into three services models known as SPI (Saas PaaS IaaS) models shown in Figure 2.6. The Figure presents different levels of details for cloud computing. They are briefly introduced in the following subsections.

- SaaS

  Software as a Service (SaaS) is considered as a distribution model over the Internet. This service allows users have access applications and services through any computer or mobile device [32].

- PaaS

  Platform as a Service (PaaS) is used for general software development. It helps cloud providers have access to a greater level of management, and control the framework from the operating system. In PaaS model, cloud providers host development tools on their infrastructures, and users have access the tools over the Internet using APIs, Web portals or gateway software [33].

- IaaS

  Infrastructure as a service (IaaS) also called Hardware as a Service (HaaS) is the highest level of customization and management offered by an IaaS provider. It helps consumers allocate their storage capacity. A large computing infrastructure and a

service based on hardware virtualization provides by Amazon Elastic Compute Cloud (EC2) [33].

2.5.2 Cloud Deployment Models

The model that describes the environment where cloud application and services installed to be available to the users identified as cloud computing deployment model [34]. Cloud computing can be grouped into four deployment models including public, private, community and hybrid cloud as depicted in Figure 2.6. Here, the four deployment models are explained in detail.

- **Public Cloud**

  Public cloud is comprised of multi-tenant dynamic resources provided over the Internet. This type of a cloud is owned by cloud providers, and relies on the standard cloud-computing model in which resources, applications, and storage operate for the benefit of public cloud consumers. An important benefit for this demographic is that no upfront cost is required for computing resources, ongoing management, and maintenance. Here, cloud providers are responsible for such costs. Examples of these services providers are Amazon EC2, Windows Azure service platform, and the Google app engine [35].

- **Private Cloud**

  Private Cloud is a type of cloud that creates a virtual environment consisting of multiple customers. In this model, organizations can prepare computing resources across different applications, departments or business units. The benefits of a private cloud are its strengths in security, control resource configuration, and data storage in addition to its convenience [34].

- **Community clouds**

  Community clouds are used by different communities of organizations that have shared concerns such as conformity or security considerations. The computing infrastructure may be provided by internal or third-party suppliers. In this model, the

communities benefit from public-cloud capabilities, but they also know their association. Therefore, they have fewer fears about security and data protection [34].

- **Hybrid cloud**

  In a Hybrid cloud, computing infrastructure is a combination of two or more cloud infrastructures, such as private, community or public. This deployment model provides a similar trust model for personal services with low cost as in a private cloud. However, having both public and private clouds working together requires interoperability and portability of both applications and data to allow communication between the models [32].

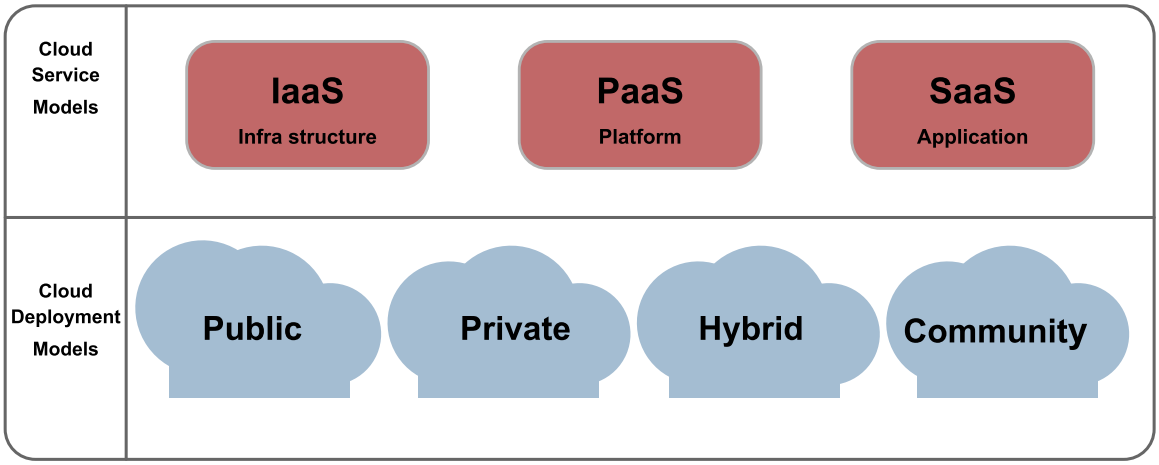

Figure 2.6: Cloud service and deployment models

2.6 Amazon Web Service

Amazon Web Services (AWS) [23] provides a full set of highly available services to deliver scalable applications over the Internet. AWS produces on-demand access to highly durable storage, low-cost computing, high-performance databases, and the tools to manage these resources with a pay-as-you-go interactive query service. It is worth mentioning that the AWS cloud goes beyond the provision of basic resources. AWS provides a high level of business agility because it allows consumers to inject agility into organization as much as

they need. Then, when they do not need those resources, consumers no longer have to pay for them. In addition, it is controllable through automation tools.

2.6.1 Amazon EC2

Amazon is the leading IaaS provider. Under the name "Amazon Web Services" they offer many services, amongst which Amazon Elastic Compute Cloud (EC2) is probably the most meaningful and the most powerful representative, one that allows users can run applications based on arbitrary platforms with different settings [36].

EC2 [25] is a web service that focuses on providing scalable computing capacity in the AWS cloud environment. It allows developers to manage virtual machines that are running on an operating system. Also, developers may use EC2 to develop and deploy applications faster and easier. Using such a service, users are able to launch as many or as few instances as they need. Moreover, it enables one to quickly scale capacity up or down as ones computing requirements change, while paying only for capacity that is actually used.

2.6.2 Amazon Auto-scaling Groups

AWS offers a replication method called Auto-Scaling Group (ASG) [37], as part of the EC2 service. This solution is part of the ASG, which consists of a set of instances that can be used for an application. ASG uses an automatic reactive approach. In ASG, a minimum and maximum number of instances for each group can be specified so that ASG is constrained to never go below or above the specified range. If you specify a desired capacity, either when you create the group or at any time afterward, ASG helps your group to maintain these many instances. If scaling policies were determined, ASG would be enabled to launch or terminate an instance upon increasing or decreasing application demands. For instance, in Figure 2.7, ASG has a minimum size of one instance, a desired capacity of two instances, and a maximum size of four instances. The scaling policies that are defined adjust the number of instances, within the user-defined minimum and

maximum number of instances, based on the criteria that are specified.

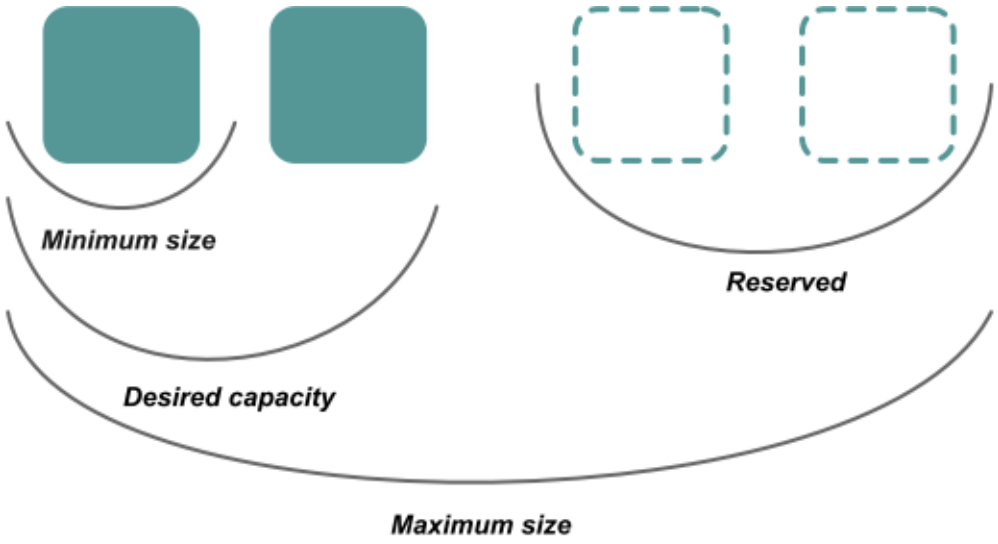

Figure 2.7: Amazon Auto-scaling Groups

2.6.3 Amazon Elastic Beanstalk

AWS Elastic Beanstalk [38] is a service for deploying and scaling web applications. It provides a platform for different programming languages including Java, PHP, Python, Ruby, Go, as well as web containers such as Tomcat, Passenger, Puma and Docker containers, with multiple configurations for each. It also provides an environment for clients to upload their code simply and automatically handles the deployment from capacity provisioning, load balancing, auto-scaling to application-health monitoring. At the same time, the users retain full control over the AWS resources powering their application and can access the underlying resources at any time without any additional charge. Users pay only for the AWS resources needed to store and run their applications.

2.6.4 Elastic Load Balancing

Elastic Load Balancing (ELB) [39] activates a load balancer that spreads load across all running instances automatically. As an example, once an instance is terminated, the load

balancer will not send requests to this instance anymore; hence, it will distribute the requests across the remaining instances. ELB also tracks and monitors the availability of an application, by checking its health periodically (i.e. every five minutes). If this check fails, AWS Elastic Beanstalk will execute further tests to detect the cause of the failure. Particularly, the existence of the load balancer and the ASG is examined and it ensures whether at least one instance is running in the ASG. Moreover, automatically distribute incoming application workloads amongst multiple EC2 instances in the cloud is the main objective of ELB.

2.6.5 Amazon CloudWatch

An auto-scaling system needs the support of a monitoring system that provides measurements of user requests. To this end, Amazon CloudWatch [40], which is part of the AWS, enables monitoring of AWS resources and the customer applications running on the Amazon infrastructure. CloudWatch automatically provides metrics for CPU utilization, latency, and request counts. Moreover, users can stipulate additional metrics to be monitored, such as memory usage, transaction volumes or error rates. The CloudWatch interface shown in Figure 2.8 provides statistics, and the monitoring can be viewed in a graph format. Additionally, users can set notifications called alarms that are set off when something is being monitored.

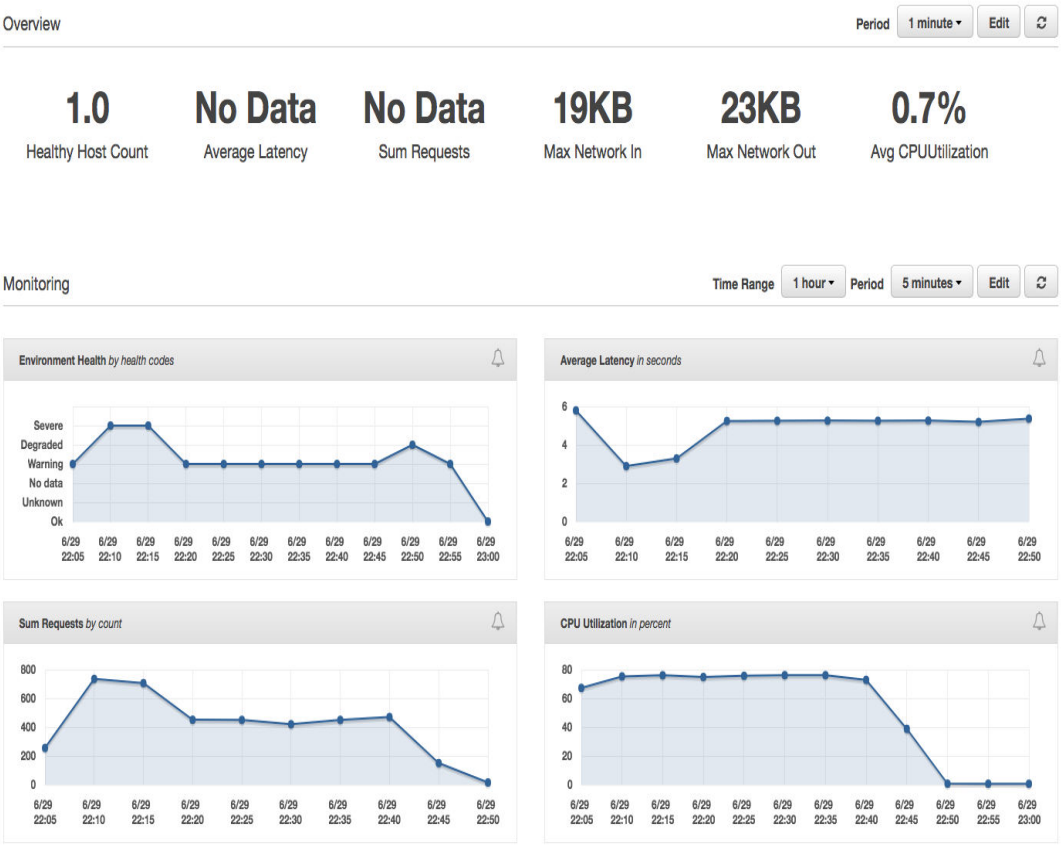

Figure 2.8: Overview of CloudWatch metric summary interface

2.7 Conclusion

In this chapter, we have presented the topics that form a contextual background for the research work carried out in this thesis. We have explained the concept of view synthesis and its components. Then, we provided a brief explanation of threads and multi-threading, which is one of the core features supported by many languages, including C++ and Java. We also introduced multi-threading programming constructs in C++ including synchronization techniques and pthread library that we used in our proposed scenarios. Finally, description of cloud computing and its services including AWS is presented in this Chapter in detail.

Chapter 3

Literature Review

3.1 Introduction

In this chapter, we will discuss the state-of-the-art of the research areas related to the core contributions of this thesis. This has been organized into two main research categories. The first category is comprised of view synthesis studies in the past few decades that have introduced many different view synthesis approaches focusing on different methods for generating a virtual view. The second category is comprised of studies related to performance evaluation of cloud-based applications, including dynamic resource provisioning approaches and dynamic auto-scaling of IaaS cloud and their impact on performance in cloud-based applications. The objective here is to introduce some related work and describe the advantages and major drawbacks of these methods.

3.2 View Synthesis

One of the earliest attempts toward virtual view synthesis is the work of Skerjanc and Liu [41] in which they synthesized an intermediate view based on visual feature matching between three camera views. To reduce the effect of occluded points, they proposed a trinocular arrangement of the cameras. Although their approach reduces occlusion artifacts, it

requires more complicated camera setup and higher computational resources accordingly. Later, Ott et al. [42] used a stereo setup to generate a virtual view in the middle of the baseline between two cameras. Stereo matching based on visual feature matching is carried out to estimate disparity map and thus recovering three-dimensional scene reconstruction. In [43], a high quality layered representation of virtual views is proposed that enables an enhanced experience by estimating and counterbalancing depth discontinuities. In their approach, a color based segmentation is also employed for more consistent correspondence matching between multiple views. In this layered approach based on color information, a more accurate representation can be achieved; however, at the cost of more computational cycles.

To further reduce this visual depth artifact, a so-called "winner-take-all" approach is introduced in [44] as an alternative to the more commonly used linear filtering approach. linear filters, this method produces less ghosting effect as the result of intensity averaging. Obviously, "winner-takes-all" filtering is not an ideal solution for real-time applications due to its computational complexity. The authors emphasized on the importance of disparity filtering rather than focusing on the estimation itself.

In the literature, a variety of research has focused on view synthesis from the standpoint of stereoscopic solutions [45][46][47]. Park et al. studied 3DTV in [47] and proposed a stereo based method for estimating disparity maps and generating synthesized view from the estimated disparity. Authors in [48] analyzed generating in-between views as a pixel-based registration and a pixel interpolation problem. To accelerate the synthesis process, authors in [49] optimized the entire synthesis pipeline for throughput using modern GPU hardware. They focused more on GPU implementation by relying on a basic stereo model for their mathematical model. Fukushima et al. [50] focused on a disparity optimization strategy called Multi-Pass Dynamic Programming (MPDP) targeting real-time rendering. Global belief propagation method with its parallel GPU implementation is proposed in [51] and claims to achieve 45x speed up over CPU implementation for real-time application.

Similarly, in [52], dynamic GPU programming is used to estimate high quality disparity image in real-time. Mori et al. [53] accelerated the synthesis pipeline for 3D FTV applica-

tion by offloading the disparity estimation as opposed to view-dependent depth estimation and using 3D warping instead. However, being inclined to a particular viewpoint rather than an arbitrary one is a downside of their optimization.

More recently, in [54], interactive view synthesis is studied from the cloud computing perspective where the authors transfer a significant amount of processing operations to the clients' end, and the main data which consists of depths and images are transmitted with the help of cloudlets. In their research, authors proposed a polynomial time algorithm for optimizing the cloudlet-synthesis navigation that minimizes a distortion penalty. Since the literature lacks extensive studies on view synthesis from the cloud computing standpoint, we present implementation approaches for the realization of arbitrary view synthesis on cloud platforms. As opposed to many existing works, ours can recover any arbitrary view between two recorded views.

Unlike [53][54] which focuses more on client-host interaction and minimization of transmission delays, we analyzed scenarios for better load balancing over cloud resources. It is also worth noting that our synthesis approach can be extended from two views to multi-view without loss of generality.

3.3 Cloud Computing (Elasticity)

In moving toward an understanding of elasticity, many researchers are focusing on the cloud-based applications ability to automatically scale based on user demands. Auto-scaling, or elasticity, is an important characteristic of cloud computing technology. It refers to the ability of a system to dynamically adapt its resources in an on-demand way over time [55]. Elasticity of cloud was addressed variously by different studies including [56][57][58]. Brebner et al. in [59] utilized three real-world applications and workloads obtained from their SOA performance-modeling approach. This study evaluated the performance of each of the application workloads, running on IaaS Amazon cloud platform, based on four scenarios. The impact of changing elasticity thresholds was evaluated for each scenario defined in Amazons auto-scaling rules. The performance of the applications was determined

in terms of a number of metrics including server cost and end-to-end response time SLA .

Ghosh et al. in [57][58] developed models focusing on end-to-end performance analysis of IaaS cloud elasticity. Their models were based on a stochastic reward net to analyze the QoS of resource provisioning and cloud-resource service requests submitted by consumers of the IaaS cloud. The effects of dynamic workload changes of various IaaS cloud consumers were quantified by provisioning response delay and service availability. A new method to support cloud consumers in measuring and comparing performance-costs of virtual cloud servers from different providers was proposed by Lenk et al. [60]. The CPU utilization of cloud servers and the application response time were the metrics used in their evaluation and are also among the primary metrics we used in our evaluation and analysis of elasticity approaches.

Dejun et al. [61] focused their evaluation on two different crucial performance properties of cloud servers provided by Amazon IaaS cloud, namely, performance stability, which measures how consistent the performance of provisioned cloud servers remains over time, and performance homogeneity, which measures the homogeneity of the performance behavior of provisioned cloud servers of the same type. Auto-scaling approach proposed by Dutta et al. [62] attempts to find the optimal combination of both horizontal and vertical scaling actions for different applications running on an IaaS cloud. Automated scaling mechanisms for applications running in IaaS cloud environments have been proposed by many research studies [63][64][65].

The auto-scaling mechanisms proposed by [64][65] were based on predefined scaling-policy rules and cloud-server configurations. This feature is an important characteristic of these auto-scaling mechanisms because it implies that they are not bounded to a specific cloud infrastructure. Furthermore, such mechanisms aim to trigger timely and efficient scaling actions based on user-defined cloud resource utilization rules.

Very recently, Wen-Hwa et al. [66] analyzed auto-scaling from the perspective of multiple heterogeneous resources strategy and proposed to overcome limitation of single-type resource provisioning. Among different publicly available auto scaling services, AWS Beanstalk service provides Infrastructure-as-a-Service (IaaS). Hector et al. [67], proposed

scaling strategy for AWS Beanstalk that dynamically adjusts a threshold to minimize the time for the creation and release of virtual machines. In our implementation, we also used AWS Beanstalk service for dynamic resource allocation.

3.4 Conclusion

This chapter briefly reviewed some of the important existing works that are relevant to the major contributions of this thesis. We presented a summary of literature studies related to view synthesis methods, from the earliest attempts toward generating a virtual view to very recent studies. Additionally, different approaches were summarized in terms of elasticity and cloud-based applications, which have been studied in recent decades. To the best of our knowledge, no other work has targeted cloud-based load balancing, scaling and parallelization for view synthesis.

Chapter 4

Proposed System

4.1 Introduction

In this chapter, we will present our proposed method, including describing the view synthesis algorithms components across multiple threads by presenting multi-threaded scenarios using 1, 2 and 4 threads and the corresponding assessments in order to obtain faster and higher quality view synthesis. We will also show our evaluation in various parallel implementation scenarios based on quantitative metrics, including speedup, efficiency and CPU utilization. Lastly, for cloud deployment, we will present our deployed application onto AWS cloud resources including AWS CloudWatch and analyze the elasticity by relying three different metrics including request count, latency and CPU utilization.

4.2 Multi-threaded view synthesis

Our view synthesis algorithm proposed by [2] as baseline that has been written in C/C++ using the open source computer vision (Open CV) library. We adapted this algorithm for parallel processing. The adapted algorithm accepts images captured from multiple cameras, along with corresponding depth information, to generate a virtual image from a unique viewpoint in order to obtain faster and higher quality view synthesis. This

approach is based on the homography calculation and relies on the provided depth maps. The estimated depth is then used in a backward direction to warp the cameras' image onto the virtual view. Finally, we use a depth-aided inpainting strategy for the rendering step to reduce the effect of disocclusion regions (holes) and to paint the missing pixels. We emphasize on data level parallelism for our implementations.

Our view synthesis algorithm has been broken down into the following blocks: homography estimation, 3D point projection, median and bilateral filtering, recovering camera poses, calculating virtual depths, and virtual image. We then selected the data independent blocks for parallel processing.

For parallelization, we used the C++ pthread library (POSIX). POSIX allowed us to distribute the computations into different CPU threads. However, one of the most fundamental issues in parallel processing is the problem of synchronization. Synchronization ensures application's critical parts not to be simultaneously executed when one's result is used by the others. Synchronization can become even more critical when multiple processors are likely to access and/or modify the shared data. Data synchronization is required to guarantee that a true copy of the data will pass through each processor's task. The method to distribute the load with data level parallelism can be grouped into two major parallel scenarios: namely, parallel threads with synchronization after each step and parallel threads with synchronization after all steps. However, at first, we need a sequential version without any parallelization shown in Figure 4.1, in which all steps are processed by a single thread to create the virtual view depth based on the given depth of right and left views.

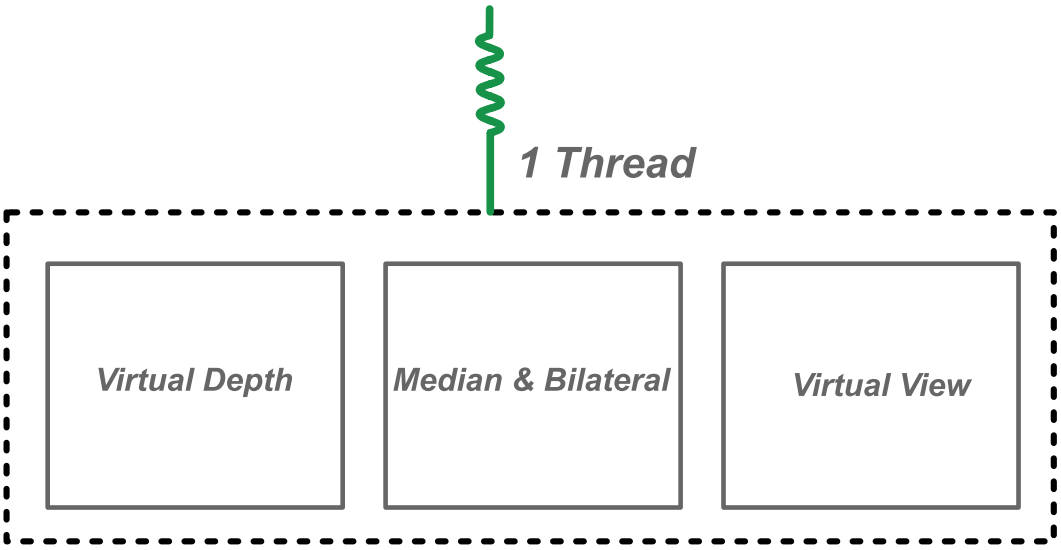

Figure 4.1: Sequential model with 1 thread

4.2.1 Thread synchronization after each step

Towards the parallel implementation, in the first scenario, we divided each image into 4 equal parts (tiles) so that each tile can be processed in separate threads. This is done by updating all possible pixels locations in the virtual depth map based on corresponding values in the left and right view. Pre-computed homography matrices are used to map pixel locations from the virtual view to both left and right views.

Graphical illustration of virtual depth estimation is shown in Figure 2.4. This computed depth then undergoes median and bilateral filters to smooth out the missing pixels (holes) in order to reduce the artifacts introduced by unseen 3D locations. The estimated depth is

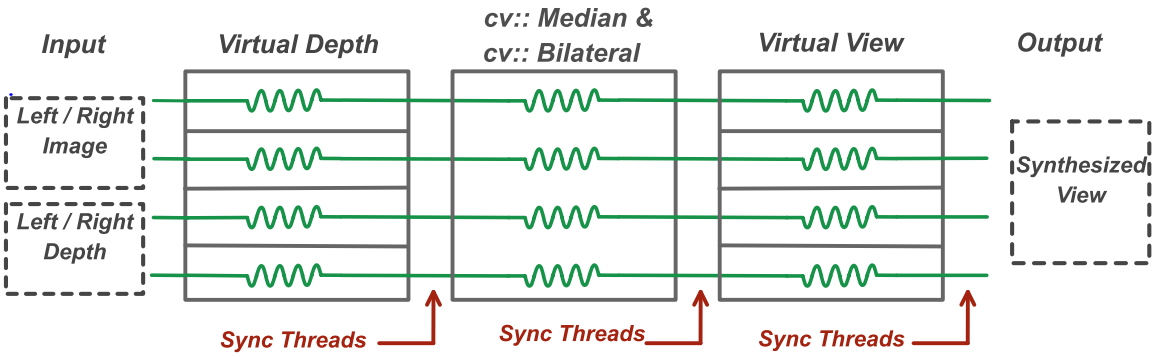

Figure 4.2: Synchronized threads after each step

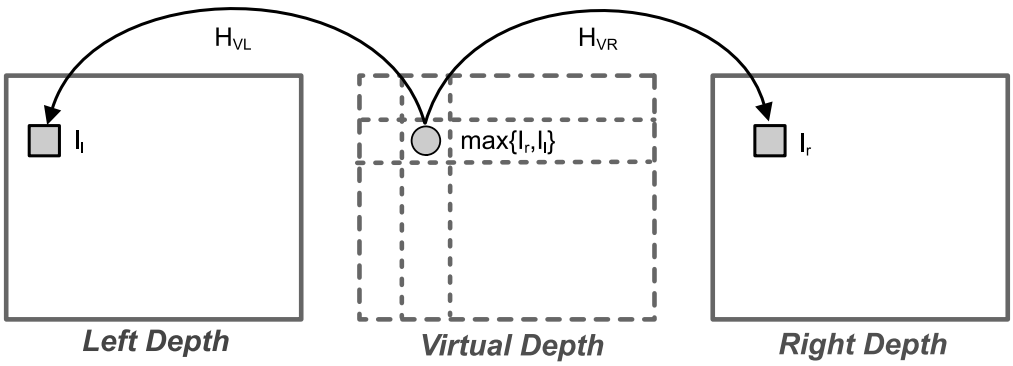

Figure 4.3: Virtual depth estimation from left and right depth maps

then used to update the virtual view pixels similar to the depth image calculation. In this scenario, all threads are synchronized at the end of each step including depth computation, median and bilateral filters calculation, and virtual image computation. Figure 4.3 visually explains this scenario.

Although it is guaranteed that in the above implementation no critical parts of the program will be executed simultaneously, it may not yield the optimal efficiency, because thread synchronization is time-consuming resulting in some threads to stay idle while others are working. This effect becomes even more significant especially when we employ a larger number of threads in a working group. One obvious solution to reduce the idle time induced by synchronization is to minimize the number of times at which threads join up. It is worth mentioning that this scenario produced identical results to the sequential version as expected.

4.2.2 Thread synchronization after all steps

In the second scenario, as shown in Figure 4.4, each thread is responsible for computing depth view, applying filters and computing virtual view sequentially by themselves. Here the goal is to minimize the total thread idle time by synchronizing the threads only once after all steps have been completed. As we show in the performance evaluation, this mechanism brings higher efficiency than the previous scenario. This efficiency comes from less waiting of yielded threads as when they needed to be synchronized between each block.

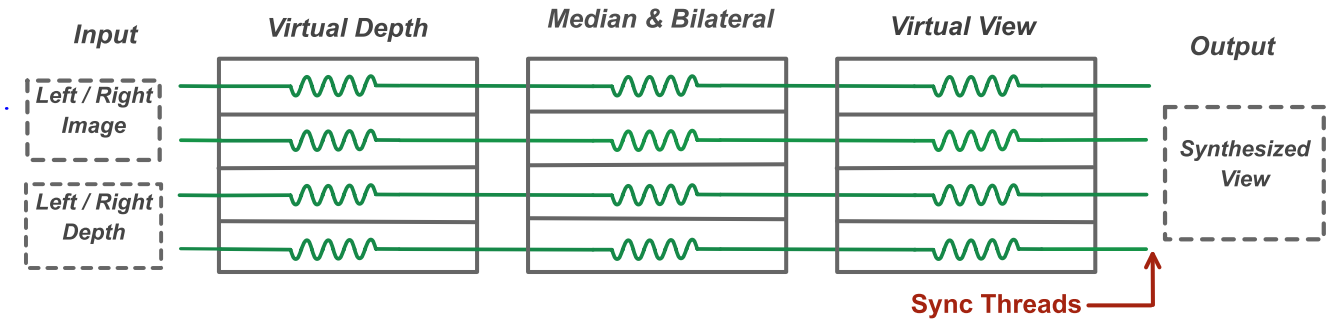

Figure 4.4: Synchronized threads once all steps are done

This means that threads are more likely to be blocked by synchronization barrier when dealing with more active threads. Since the threads are synchronized once all three steps have been completed, the threads' idle time is reduced.

One drawback of the second scenario is that it might introduce small artifacts at the boundaries of each thread's region of interest due to the data race condition is shown in Figure 4.5. As we can see, the red pixel in T1 region (blue) has been mapped to T2 region (green) inducing a small seam. However, we have observed that if we apply Median and Bilateral filters, then this artifact is barely visible. We have presented an example of such artifact in Figure 4.6 for illustration purpose that left shows seam artifacts induced by single synchronization and in right one median and bilateral filters compensate the seams.

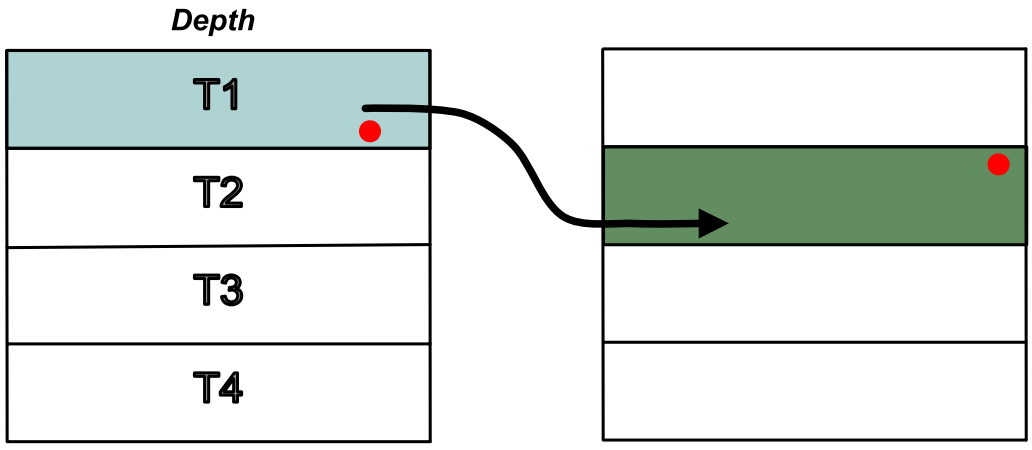

Figure 4.5: Small seam at boundaries of thread's region

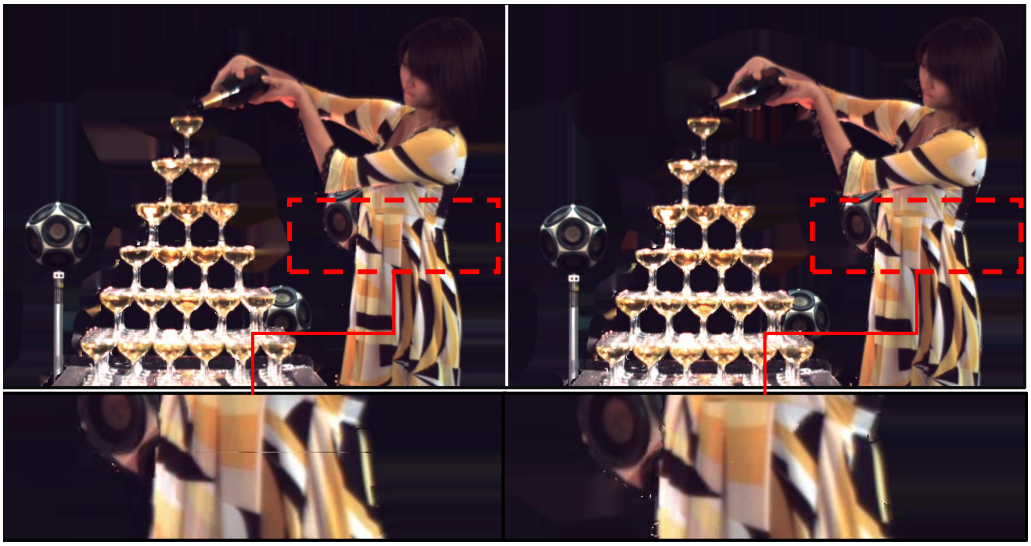

Figure 4.6: Seam artifacts induced by single synchronization

4.3 Multi-Thread Performance Metrics

The tremendous growth in production and advances of computer hardware with multi-processor units has turned parallel computing a hot topic in the past decade [68]. As a consequence, performance evaluation of systems exploiting such hardware resources has become incidentally an important topic.

To evaluate our proposed approach for multi-threaded view synthesis, we relied on performance metrics like speedup, efficiency, and utilization which are commonly used in parallel computing [69]. These metrics are briefly introduced in the following subsections.

4.3.1 Speedup

Speed up is used to measure how many times faster a parallel application outperforms its sequential version. It can be expressed by:

$$S(p) = \frac{T(I)}{T(p)}, \tag{4.1}$$

where S(p) is unitless speedup, T(1) the execution time using one processor and T(p) the execution time using p processors.

4.3.2 Efficiency

Efficiency measures the effectiveness of a parallel implementation. In another word, efficiency quantizes the factor of speedup achieved with p processors. In practice, it is usually impossible to achieve full efficiency since, there are always hidden costs associated with using parallel or threaded hardware. The efficiency is formulated as:

$$E(p) = \frac{S(p)}{(p)} = \frac{T(I)}{pT(p)}, \tag{4.2}$$

where E(p) is efficiency, p is number of processors, S(p) is speedup.

4.3.3 Utilization

Another metric to measure the effectiveness of a parallel application is processor's utilization. It refers to the maximum usage of computing resources of computers processors such as CPUs and GPUs. It is reported as the fraction of used computational resources (instructions) over maximum potential resources that the processors could give.

$$U(p) = \frac{W(p)}{pT(p)}, \tag{4.3}$$

where W(p) is the total number of unit instructions performed by p processors. And T(p) is execution time using p processors.

4.4 View synthesis cloud deployment

The way that computer resources are allocated, and the possibility to build a fully scalable server setup on the cloud is totally revolutionized by cloud computing. Now, we have the ability to launch additional computer resources on-demand. If any application needs more

computing power, we can use them on the cloud as long as the application requires and then terminate these tasks when they are no longer needed.

Auto-scaling, or elasticity, is one important feature of cloud computing technology; it is a dynamic property of a cloud system and reflects its ability to adapt its resources in an on-demand way from the perspective of an operational system. There are several definitions of elasticity, of which one concise definition is by [70].

"Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible".

This allows clients to request and quickly receive as many resources as needed. In other words, auto scaling is a term referred to the capability of a platform on which a service or application is deployed to automatically allocate or deallocate computing resources based on the service/application needs. This empowers the application to subsequently handle increased load by adding more resources to the application while reducing the cost by down-scaling during the off-peak hours. In our work, we enabled this capability to the AWS platform through Elastic Beanstalk Group [38]. AWS Beanstalk adds another layer of abstraction to the server OS that provides various services including Auto-scaler group. We then defined auto-scaling policy to enable automatic scaling based on monitoring metrics such as count of requests, CPU utilization, or latency. Therefore, a monitoring quantity (e.g. latency) is constantly being monitored to trigger the auto-scaler and add a reserved instance or instances if the average amount reaches a pre-set threshold within a predefined breach duration (1-5 minutes), or they will be removed once the average amount hits the low of another (lower) threshold.

Then we analyzed elasticity of our deployed application and assessed its reliability using the AWS CloudWatch service. AWS CloudWatch provides a comprehensive set of monitoring metrics which are used to collect and track system-wide visibility into cloud resources, application health and performance. We relied on three main metrics to evaluate elasticity of our application and its ability to scale with changes in workload. Percentage of CPU utilization, latency time and the number of HTTP requests are primarily used

to trigger the auto-scaler and track application health and performance. Here we briefly review the three main metrics provided by Amazon CloudWatch, which we used to evaluate elasticity of our application.

Request Count

It is defined as the number of requests received by the deployed application on the server during a specific period of time. This metric which can be reported by the average, minimum, maximum or total number of requests, is a simple yet concise quantity to illustrate the traffic behind the application.

Latency

Latency measures the waiting time that clients experience during the time a request is sent until the corresponding service is received.

CPU Utilization

It refers to the percentage of processing power handed by CPU resources to execute a particular task. In AWS CloudWatch it can be reported by average, minimum and maximum percentage of CPU usage of a EC2 instance.

4.5 Conclusion

In this chapter, we described possible strategies for balancing the computational load across multiple processing units based on two different multi-thread scenarios using 1, 2 and 4 threads. We evaluated various parallel implementation scenarios relied on multi-threaded performance metrics, like speedup, efficiency and CPU utilization. In order to analyze auto-scaling and elasticity, we presented our deployed view synthesis application into AWS cloud resources based on three main metrics.

Chapter 5

Experimental Results

5.1 Introduction

This chapter will present and compare our experimental results supporting the multi-threaded view synthesis implementation. In this experiment, timings and performance metrics are measured using 1, 2 and 4 threads and the performance of the threaded implementation is assessed and compared against the scalar one. Afterwards, we will present elasticity on the cloud. For elasticity, AWS CloudWatch service is used to collect and track system-wide visibility into cloud resources, application health and performance base on percentage of request count, latency and CPU utilization.

5.2 Performance Evaluation

To assess our proposed approach for multi-threaded view synthesis, we relied on performance metrics including speedup, efficiency, and CPU utilization which are commonly used in parallel computing applications [69]. The test machine that we used is an "x86 Linux" machine with 4 physical "2.4 Intel Xeon" processors each with one thread which is deployed on the Amazon AWS cloud platform. We used "Valgrind" [38], which consists of a set of multi-purpose profiling tools with cycle estimation and memory debugging capabilities, to profile the proposed implementation.

To evaluate different strategies, we selected 4 video sequences data sets from [40]. These video sequences have the necessary depth maps and camera positions of several views which are essential for view synthesis. As can be seen from the second column in Table 5.1, we identified parallelizable parts of the view synthesis algorithm. The third column shows each part's contribution to the whole algorithm based on the number of instructions reported as percentage. In column 4 (and its sub-columns), we measured the execution time of each part in milliseconds for 1, 2 and 4 threads. We also presented the total number of instruction fetches for each step in column 5. Taking these numbers into account, we can infer an even computational balance across all exploited threads. Column 6-8 represent evaluation and performance metrics for three various numbers of threads.

By comparing the timings of 1, 2 and 4 threads, one can conclude that exploiting 4 threads gives around 1.6 - 1.7x speedup for the parallelized parts yielding 1.10 speedup of the whole algorithm. This can be explained by the extra overhead associated with initializing and running multiple threads. Moreover, since primary parts do not contribute 100% of the whole algorithm, it is not expected to achieve 4x speedup in overall. This can be also verified by the efficiency numbers and utilization values that do not exhibit 100% efficient realization.

We repeated the same experiment for the second scenario and reported the results in Table 5.2. Again, we can see that we gained a smaller speedup using 2 threads compared to 4 threads clearly due to the smaller number of threads. However, the efficiency of 2 threads implementation is higher due to the smaller CPU overhead associated to creation and synchronization of 2 threads. In fact, more efficient execution is achieved by minimizing the number of synchronizations.

By referring to Table 5.1 and Table 5.2, we should mention that in our performance results, the timings and performance metrics are measured using 1, 2 and 4 threads. And all reported timings are averaged over a total of 10 frames. Since filtering, virtual depth and image computation are merged in the second scenario shown in Table 5.2 executed over several threads, corresponding sequential measurements are not available(noted by N/A).

Table 5.1: Performance result of the first scenario for four sequences

# of threads / Dataset		Percentage	Time(ms) - O3			Instruction fetches			Speedup		Efficiency		Utilization	
			1	2	4	1	2	4	2	4	2	4	2	4
Balloons 1024×768 	main()	100	1068.2	1001.1	993.89	2005479270	1806882207	1806882137	1.067	1.0748	0.5335	0.2687	0.5921	0.298
	↳ viewsynthesis()	91.87	994.4	950.6	942.8	1794794358	1596197201	1596197226	1.0461	1.0547	0.5231	0.2637	0.5882	0.297
	↳ applyFilters()	76.83	625.2	610.2	615.3	7692396	7687170	7687375	1.0246	1.02	0.5123	0.255	0.5126	0.255
	↳ calcVirtualDepth()	3.47	27.7	16.7	16.4	90486663	92060777	92079439	1.66	1.69	0.83	0.42	0.8158	0.413
	↳ calcVirtualImage()	5.24	31.7	18.9	10.6	109348098	115639352	115662873	1.68	2.99	0.84	0.75	0.7943	0.709
Kendo 1024×768 	main()	100	812.5	765.2	755.2	526614168	327947418	327946511	1.0618	1.0759	0.5309	0.269	0.8525	0.432
	↳ viewsynthesis()	93.41	732.4	715	704.9	315882286	117215448	117214492	1.0243	1.039	0.5122	0.2598	1.3803	0.7
	↳ applyFilters()	82.87	651.1	657.1	663	7692443	7687221	7687375	0.9909	0.9821	0.4955	0.2455	0.4958	0.246
	↳ calcVirtualDepth()	2.81	27.7	17.1	9.5	90486681	92060777	92079439	1.62	2.92	0.81	0.73	0.7962	0.717
	↳ calcVirtualImage()	4.24	31.8	19.1	10.6	109348014	115639352	115662380	1.66	3	0.83	0.75	0.7848	0.709
Pantomime 1280x960 	main()	100	1179	1111.6	1067.7	800252184	489826615	489827318	1.06	1.1	0.53	0.275	0.8659	0.449
	↳ viewsynthesis()	92.58	1053.7	1032.9	989	470108834	159683090	159684050	1.02	1.07	0.51	0.2675	1.5014	0.788
	↳ applyFilters()	81.16	932.2	949.7	929.6	11582650	11577224	11577334	0.98	1	0.49	0.25	0.4902	0.25
	↳ calcVirtualDepth()	3.16	44	25.3	14.13	141360791	143819945	143838607	1.74	3.11	0.87	0.78	0.8551	0.767
	↳ calcVirtualImage()	4.76	49	29.3	16.4	170480631	180670281	143838607	1.67	2.99	0.84	0.75	0.7926	0.889
Champagne1280x960 	main()	100	4463.1	4439.7	4410	20769029825	20458604027	1806989113	1.01	1.01	0.505	0.2525	0.5127	2.902
	↳ viewsynthesis()	91.22	4335.1	4355.7	4330.7	20438849409	20128423604	1596197226	1	1	0.5	0.25	0.5077	3.201
	↳ applyFilters()	81.2	930.1	928.5	923.2	11582543	11577534	7687375	1	1.01	0.5	0.2525	0.5002	0.38
	↳ calcVirtualDepth()	3.01	43.61	28	22.4	141360739	180648098	180695050	1.56	1.95	0.78	0.49	0.6104	0.383
	↳ calcVirtualImage()	4.98	48.5	28.6	16	170840522	143819945	143838607	1.7	3.03	0.85	0.76	1.0097	0.903

Table 5.2: Performance result of the second scenario for four sequences

# of threads / Dataset		Percentage	Time(ms) - O3			Instruction fetches						Speedup		Efficiency		Utilization	
			1	2	4	1	2		4			2	4	2	4	2	4
Balloons 1024×768 	main()	100	1068.2	988.9	956.1	2005479270	2058035696		1806882137			1.0802	1.1172	0.5401	0.2793	0.5263	0.31
	↳ viewsynthesis()	91.87	994.4	938.7	907.3	1794794358	1847350789		1596197226			1.0593	1.096	0.5297	0.274	0.5146	0.308
	↳ calcVirtualDepthAndImage()	N/A	N/A	833.7	693.6	N/A	221330997	110366792	221360031	54492376	55623637	N/A	N/A	N/A	N/A	N/A	N/A
								114309006		55621490	57711457						
Kendo 1024×768 	main()	100	812.5	765	754.2	526614168	327947418		327946511			1.0621	1.0773	0.5311	0.2693	0.8528	0.432
	↳ viewsynthesis()	93.41	732.4	714.7	698.1	315882286	113665763		117214492			1.0248	1.0491	0.5124	0.2623	1.424	0.707
	↳ calcVirtualDepthAndImage()	N/A	N/A	690.1	641.4	N/A	221330997	110964205	221361069	54471440	55623637	N/A	N/A	N/A	N/A	N/A	N/A
								110366792		55643477	55562515						
Pantomime 1280x960 	main()	100	1179	1051.1	1027.7	800252184	483934861		484829994			1.12	1.15	0.56	0.2875	0.926	0.475
	↳ viewsynthesis()	92.58	1053.7	988.9	971.2	470108834	153791496		154686453			1.07	1.08	0.535	0.27	1.6354	0.821
	↳ calcVirtualDepthAndImage()	N/A	N/A	878.4	876.6	N/A	345793934	172562281	345823993	85392279	86811373	N/A	N/A	N/A	N/A	N/A	N/A
								173267653		86689780	86689780						
Champagne1280x960 	main()	100	4463.1	4429.3	4401.9	20769029825	20479960116		20521592402			1.01	1.01	0.505	0.2525	0.5121	0.256
	↳ viewsynthesis()	91.22	4335.1	4319.7	4310.7	20438849409	20149779664		20191411796			1	1.01	0.5	0.2525	0.5072	0.256
	↳ calcVirtualDepthAndImage()	N/A	N/A	3981.2	3777.3	N/A	345793934	172526281	345823819	85371345	86832133	N/A	N/A	N/A	N/A	N/A	N/A
								173267653		86810177	86810164						

From the graph presented in Figure 5.1, one can perceive better comparison between each individual scenario for the overall algorithm speedup. It is clearly visible that scenario 2 outperforms scenario 1 especially for sequence Balloons and Pantomime. Since parallelizable parts of sequence Champagne only contribute minor portion of the whole algorithm, the speedup difference between scenarios is negligible. We can conclude from speedup values that we can achieve higher speedup with scenario 2 even using smaller number of threads (as seen for Pantomime).

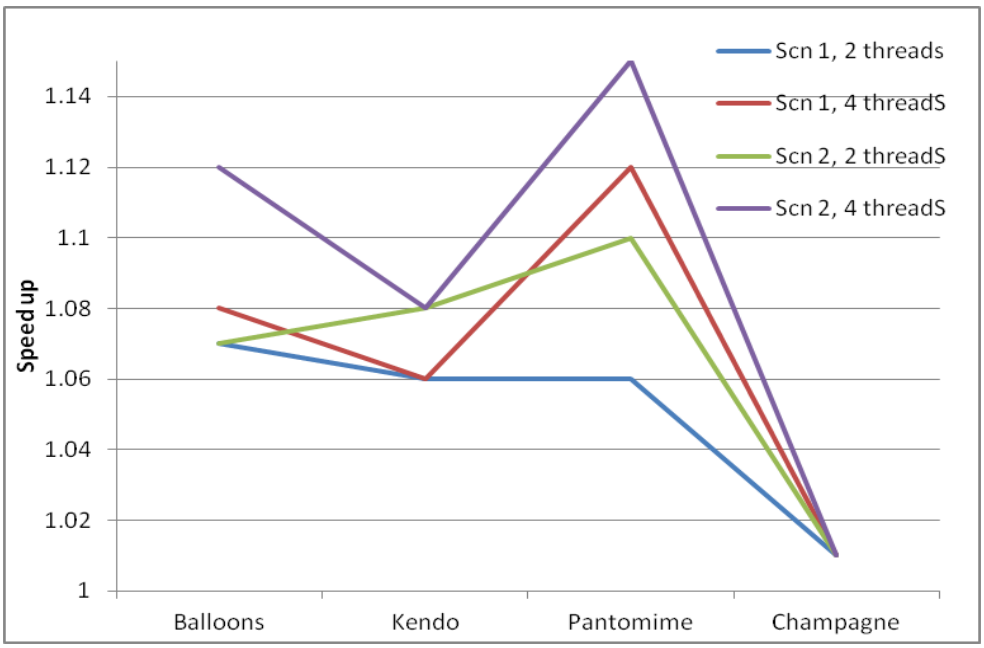

Figure 5.1: Speedup for the four different sequences

Similar to Figure 5.1, efficiency and utilization measures are presented in Figure 5.2 and 5.3 respectively. By referring to efficiency values, we can see that using two threads increases efficiency significantly while scenarios have no big advantages over each other. In other words, there is small advantages for scenario 2 over 1.

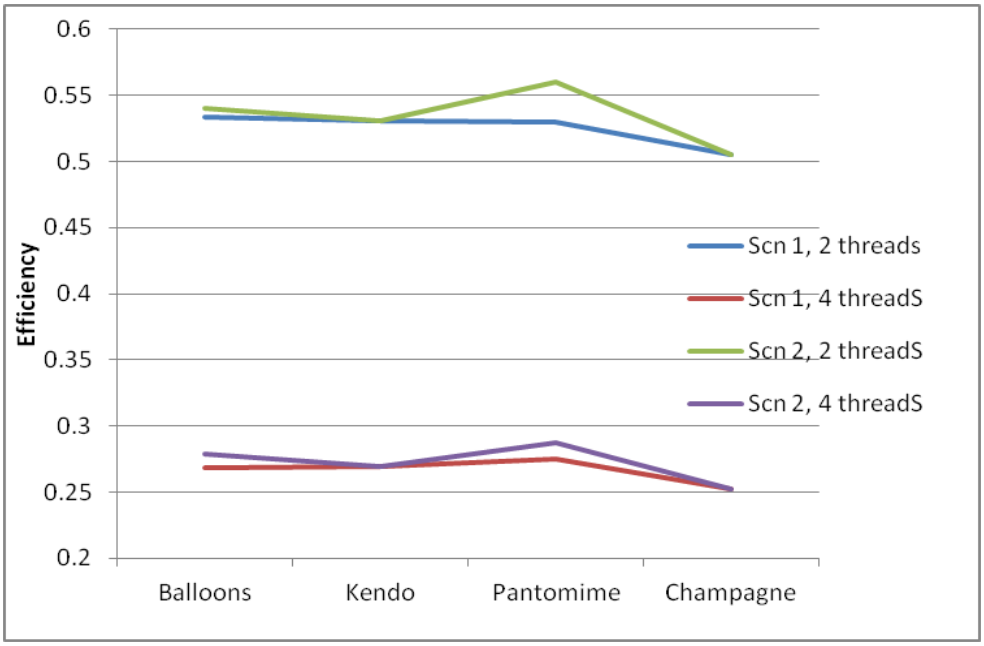

Figure 5.2: Efficiency for the four different sequences

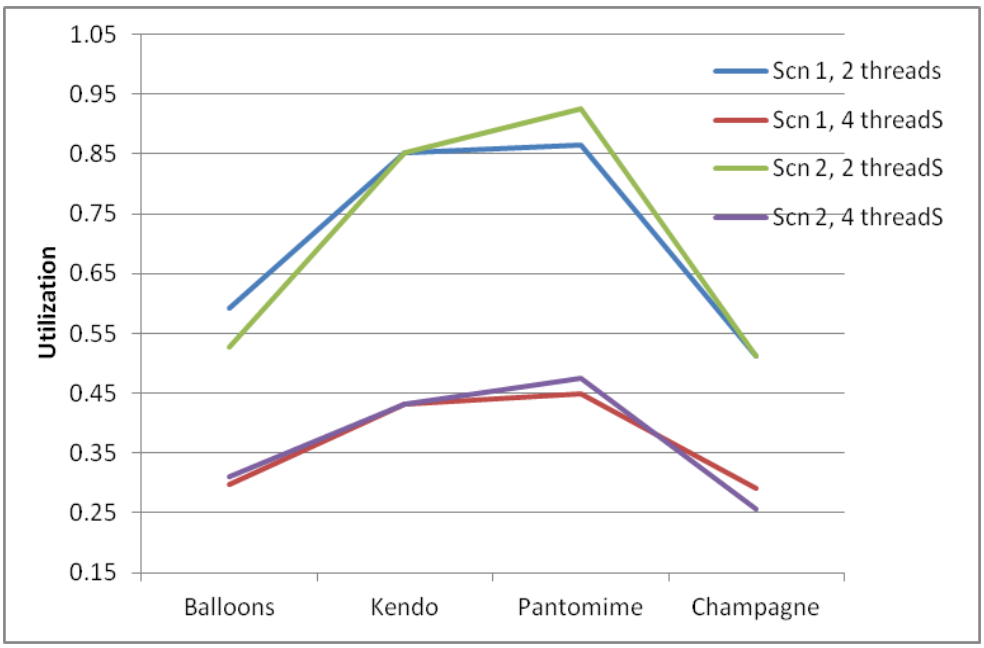

Figure 5.3: CPU utilization for the four different sequences

5.3 Auto-scaling

In this section, we present experimental result of our cloud application and analyze our findings in order to assess and validate the efficiency and reliability of our deployed application, we employed metrics provided by the same cloud service we used for deployment. The AWS CloudWatch service which provides a comprehensive set of monitoring metrics is used to collect and track system-wide visibility into cloud resources, application health and performance. We relied on three main metrics to evaluate elasticity of our application and its ability to scale with changes in workload. Percentage of CPU utilization, latency time and number of HTTP requests are primarily used to track application health and performance.

For validating the ability to scale the application, we configured auto-scaler based on percentage of CPU utilization and illustrate its monitoring parameters in Table 5.3. In Table 5.3, the first and second row show the minimum and maximum allowable number of instances for the auto-scaling group to maintain which is 1 and 4 respectively. The third row shows the selected statistic measure on which triggering is based upon. In our

experiment, the preferred statistic for CPU utilization is average. We also collected our results by choosing sum of request counts for demand evaluation. In the fourth row, we indicate the time interval between each metric evaluation. For more precise evaluation, we set all monitoring time intervals to the minimum of 1 minute. For an elastic application, it is critical to detect and discard possible short-lived momentary threshold breach. To this end, we defined the breach duration to 1 minute in row 5-6 which indicates the minimum time duration to see if a breach has occurred. Also, we defined the upper and lower thresholds for CPU utilization 75% and 65% for upper and lower bounds respectively as seen in rows 7-8.

Table 5.3: Auto-scaling policy parameters

Parameters	**CPU Utilization**
Minimum number of instances	1
Maximum number of instances	4
Trigger statistic	Average
Measurement period	1 min
Breach duration	1 min
Trigger upper threshold	75%
Trigger lower threshold	65%

To track system behavior and scaling ability to increment and decrement instances, initially, we started by sending bulk requests with a frequency of 1 request per second for a continuous period of 5 minutes. By progressively increasing the frequency to 50 requests per second in the same period of time, the workload incidentally increased until reaching the peak of 100 requests per second. That being said, the auto-scaler is responsible to increment or decrement as many instances as necessary based on the policy we defined. At this point, we continue sending requests in the same manner but with a decreasing frequency. Here we expect the auto-scaler to start to remove instances based on lower breach scale as the work-load decreases. Figure 5.4 shows the user request profile that we sent and received during the experiment. The top graph shows successfully received HTTP request at the server side. The bottom shows actual request profile by its frequency.

As pointed out, in this research we relied on three main metrics provided by Amazon CloudWatch to evaluate elasticity of our application by scaling triggered by request count

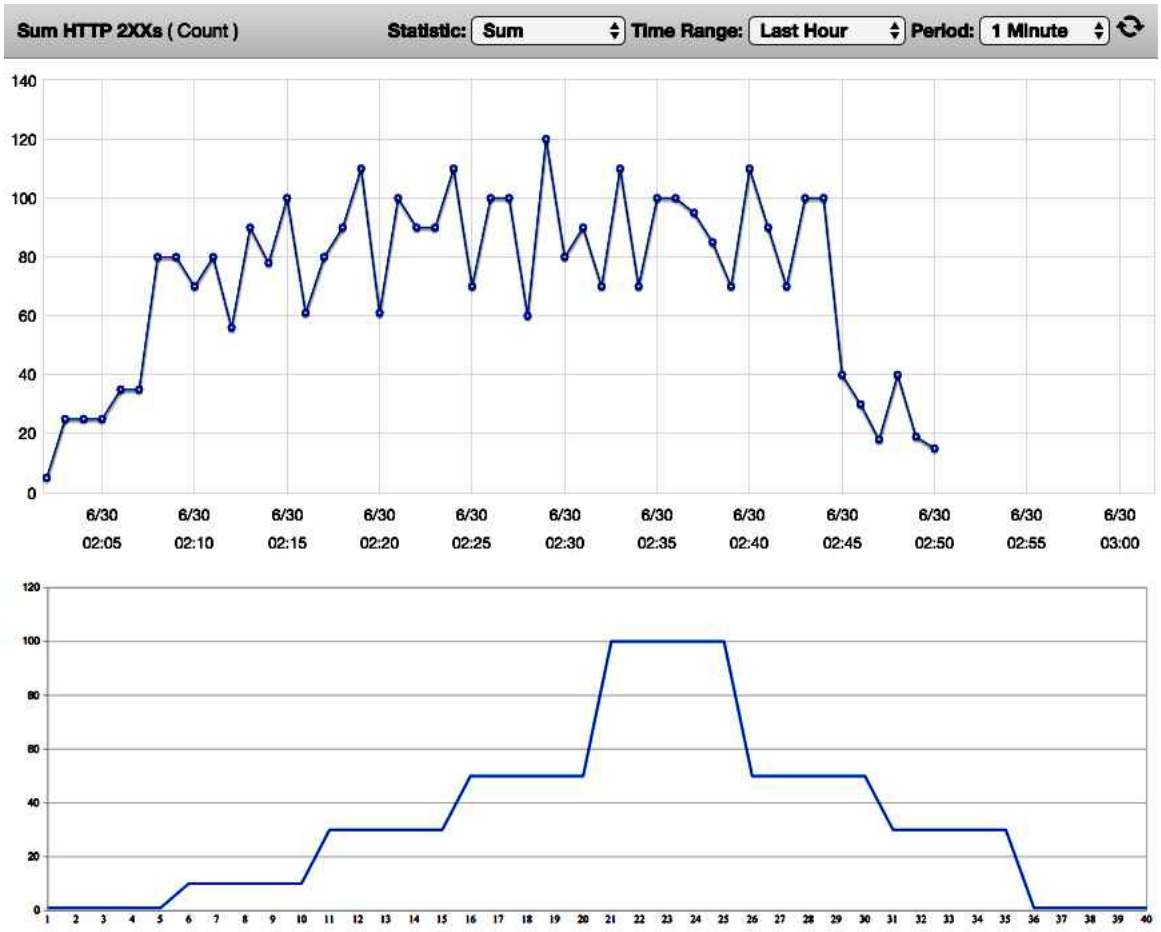

Figure 5.4: Server side successfully received HTTP request and actual request profile by its frequency

(also called demand), latency and CPU utilization. It is worth mentioning that in our work, users are able to send HTTP request (through HTML form) to the AWS Beanstalk service. To better illustrate the detailed behavior of instances to increment and decrement in terms of performing the scaling activity, initially, we started by sending bulk requests with a frequency of 1 request per second for a continuous period of 5 minutes. By progressively increasing the frequency to 50 requests per second in the same period of time, the workload incidentally increased until reaching the peak of 100 request per second. That being said, the auto-scaler is responsible to increment /decrement as many instances as necessary based on the policy we defined. At this point, we continue sending requests in the same manner but with a decreasing frequency every 5 minutes. Here we expect the auto-scaler start to remove instances based on lower breach scale as the workload decreases.

By referring to Figure 5.5, it is noteworthy that the HTTP request profile will not yield the same profile on the server side due to the network latency, possible HTTP client errors (4xx) or server errors (5xx). This becomes even more evident in the server side when it starts adding or removing instances.

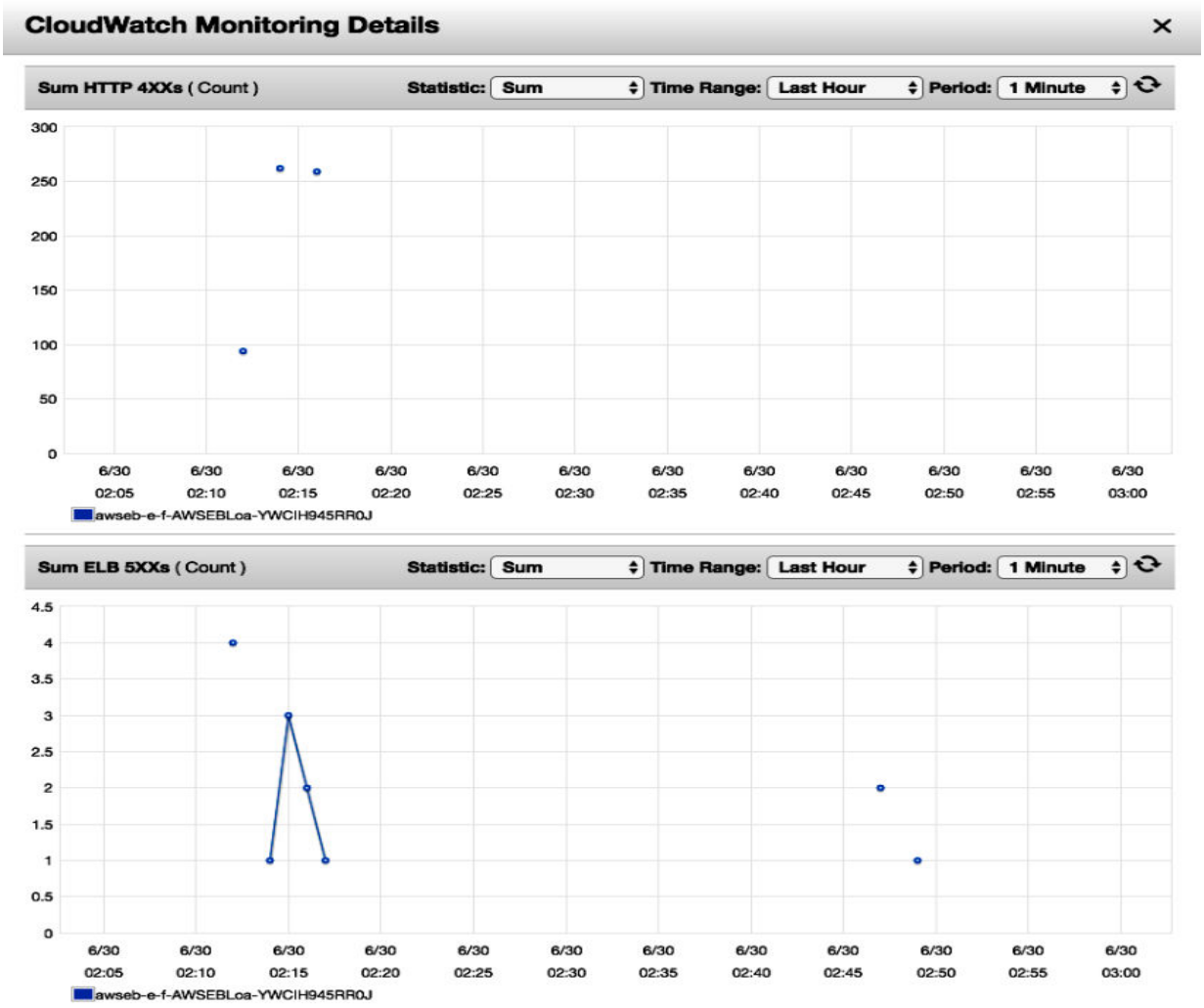

Figure 5.5: HTTP client errors and server errors

Figure 5.6, shows how the elasticity controller maintains CPU usage at a reasonable level during the experiment. In Figure 5.6, orange line shows the percentage of CPU utilization for the duration of experiment and blue shows number of in-service instances. Since we set triggering policy to launch a new instance every time CPU usage exceeds 75%, we can clearly see that by increasing the number of requests (load), the scaling policy cools down CPU usage by adding new instances. We can similarly see from the right end of the

plot that in-service instances are being terminated by decrease in CPU utilization as the result of decreasing demand.

In Figure 5.7, showing the same results with auto-scaling disabled, one can conclude that this behavior is a result of the auto-scaling service. Indeed, auto-scaler is responsible for adding/terminating instances to maintain CPU usage below the target value yet keeping the cost of resources as low as possible.

Figure 5.6: System performance with auto-scaling enabled

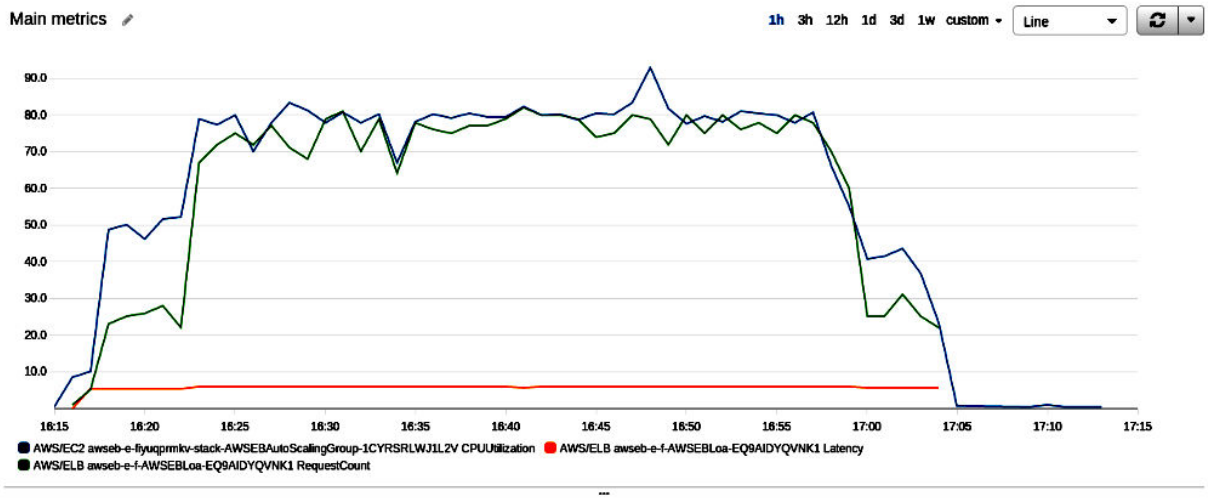

Figure 5.7: System performance with auto-scaling disabled

5.4 Conclusion

This chapter summarized the results of our experiments. We presented our experimental results on the multi-threaded view synthesis implementation and compared them against those of the sequential version. We evaluated the performance metrics for three numbers of threads (1, 2, and 4) and this was also graphically presented to facilitate a better comparison between each individual scenario. Then, we presented our findings in order to assess and validate the efficiency and reliability of our deployed application by sending bulk requests with a frequency of one request per second for a continuous period of five minutes. Lastly, we employed metrics including: number of requests, latency and utilization. These were provided by AWS CloudWatch service for the deployment our application.

Chapter 6

Conclusion and Future Work

In this thesis, fundamental steps toward analyzing the problem of novel view synthesis in a multi-camera system are investigated. These advances are aimed at realizing an efficient view synthesis algorithm that can be parallelized over multiple processing units. The main advantage of our proposed algorithm is a growing demand for fast and efficient view synthesis. In this particular direction, we present a study on parallel implementation of such a system in a consumer-based environment suitable for execution on cloud platforms. To this end, we have shown that a similar accuracy can be achieved for arbitrary view-generation using any number of camera views on a multi-core cloud platform.

The present work has focused on a cloud-based implementation for view synthesis of multi-view video using homography estimation and view warping. We presented different strategies to balance the computational load and reduce threads idle time by minimizing the number of points. We studied the increase in speed, efficiency, and improvement in resource utilization that result from parallel cloud implementation versus a sequential approach. This was done using two different solutions in terms of the way the synchronization takes place. Experimental results show an encouraging speedup ratio and resource utilization improvement. We also studied the auto-scaling features provided by the cloud services by measuring the auto-scaling performance of the solution.

As a consequence of this work, we employed a depth map of the scene, from the cameras viewpoint, to estimate depth information, conceived from the virtual viewpoint, for

generating arbitrary view. Afterwards, the estimated depth is then used in a backward direction to warp the cameras image onto the virtual view. In this case, for the rendering step, a depth-aided inpainting strategy is exploited to reduce the effect of disocclusion regions (holes) and to paint the missing pixels.

In order to have a fast and efficient view synthesis application, we present a study on parallel implementation of such a system. Our multi-threaded implementation uses the computational capacity of modern CPUs to properly balance the algorithms components across limited resources. We have shown that it is possible to achieve an accuracy comparable to the one obtained without optimization. Our experimental results have shown that thread synchronization plays an important role in efficient parallelization of the view synthesis pipeline. Relying on the results of the thesis, we aimed to reduce threads idle time by minimizing the number of points where the threads could join.

In order to deploy our application on the cloud, we added a level of on-demand adaptation through the ASG that enables a more elastic experience on the cloud. This was done in order to assess and validate our design through constant monitoring of server loads and metrics. The efficiency and reliability of our deployed application is based on three main metrics, which allow us to evaluate the elasticity: percentage of CPU utilization, latency and number of HTTP requests. The following section provides directions for future work.

In this work, we did research on a single server with four cores to implement the algorithm. In order to improve our proposed system, performance tests must be conducted using multiple servers and more fine-grained parallelism. We can also focus on bringing higher level of granularity by carrying out data and instruction level parallelism. We should expand the scope of parallelization to include other portions of the algorithm. This will also require extended profiling procedure. To this end, for future work, we can try to parallelize other parts of the algorithm by handling their data dependency and efficient memory management.

References

[1] M. Tanimoto, "Ftv: Free-viewpoint television," *Signal Processing: Image Communication*, 2012.

[2] [Online]. Available: http://www.fujii.nuee.nagoya-u.ac.jp/multiview-data/mpeg2/VS.htm

[3] G. Lafruit and et.al., "New visual coding exploration in mpeg: Super-multiview and free navigation in free viewpoint tv," *Electron. Imaging*, no. 5, 2016.

[4] W.-C. Chen, J.-Y. Bouguet, M. H. Chu, and R. Grzeszczuk, "Light field mapping: efficient representation and hardware rendering of surface light fields," *ACM Transactions on Graphics (TOG)*, 2002.

[5] T. Fujii and M. Tanimoto, "A real-time ray-space acquisition system," in *Electronic Imaging*, 2004.

[6] D. Miao, W. Zhu, C. Luo, and C. W. Chen, "Resource allocation for cloud-based free viewpoint video rendering for mobile phones," in *Proceedings of the 19th ACM international conference on Multimedia*, 2011.

[7] W. Zhu, C. Luo, J. Wang, and S. Li, "Multimedia cloud computing," *IEEE Signal Processing Magazine*, 2011.

[8] L. Yang, M. P. Tehrani, T. Fujii, and M. Tanimoto, "High-quality virtual view synthesis in 3dtv and ftv," *3D Research*, 2011.

[9] P. E. Debevec, C. J. Taylor, and J. Malik, "Modeling and rendering architecture from photographs: A hybrid geometry-and image-based approach," 1996.

[10] P. LUO, "Accurate measurement of three-dimensional deformations in de-formable and rigid bodies using computer vision," *Exp. Mech.*, 1993.

[11] S. Leutenegger, M. Chli, and R. Y. Siegwart, "Brisk: Binary robust invariant scalable keypoints," in *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2011.

[12] H. Bazargani, E. Omidi, and H. A. Talebi, "Adaptive extended kalman filter for asynchronous shuttering error of stereo vision localization," in *Robotics and Biomimetics (ROBIO)*, 2012.

[13] H. Bazargani, "Real-time recognition of planar targets on mobile devices," Master's thesis, The University of Ottawa, 2014.

[14] S.-T. Na, K.-J. Oh, C. Lee, and Y.-S. Ho, "Multi-view depth video coding using depth view synthesis," in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*. IEEE, 2008.

[15] J. Gautier, O. Le Meur, and C. Guillemot, "Depth-based image completion for view synthesis," in *IEEE 3DTV Conference: The True Vision-capture, Transmission and Display of 3D Video (3DTV-CON)*, 2011.

[16] X. Xu, L.-M. Po, C.-H. Cheung, L. Feng, K.-H. Ng, and K.-W. Cheung, "Depth-aided exemplar-based hole filling for dibr view synthesis," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013.

[17] K.-J. Oh, S. Yea, and Y.-S. Ho, "Hole filling method using depth based in-painting for view synthesis in free viewpoint television and 3-d video," in *Picture Coding Symposium, 2009. PCS 2009*. IEEE, 2009.

[18] I. Daribo and B. Pesquet-Popescu, "Depth-aided image inpainting for novel view synthesis," in *Multimedia Signal Processing (MMSP), 2010 IEEE International Workshop on*, 2010.

[19] Y. Mao, G. Cheung, A. Ortega, and Y. Ji, "Expansion hole filling in depth-image-based rendering using graph-based interpolation." ICASSP, 2013.

[20] [Online]. Available: www.buyya.com/java/Chapter14.pdf

[21] H. Kasim, V. March, R. Zhang, and S. See, "Survey on parallel programming model," in *IFIP International Conference on Network and Parallel Computing*, 2008.

[22] [Online]. Available: https://www.tutorialspoint.com

[23] [Online]. Available: https://www.techopedia.com

[24] [Online]. Available: https://https://www.techopedia.com

[25] [Online]. Available: https://https://support.microsoft.com

[26] [Online]. Available: http://www.embeddedlinux.org

[27] P. Mell, T. Grance *et al.*, "The nist definition of cloud computing," 2011.

[28] M. A Vouk, "Cloud computing–issues, research and implementations," *CIT. Journal of Computing and Information Technology*, vol. 16, no. 4, 2008.

[29] [Online]. Available: http://https://apprenda.com/library/cloud/

[30] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, 2008.

[31] [Online]. Available: http://queue.acm.org/detail.cfm?id=1554608

[32] P. Pouladzadeh, "A cloud-assisted mobile food recognition system," Ph.D. dissertation, University of Ottawa, 2017.

[33] A. Shawish and M. Salama, "Cloud computing: paradigms and technologies," in *Inter-cooperative collective intelligence: Techniques and applications*, 2014.

[34] M. Hamdaqa, "An integrated modeling framework for managing the deployment and operation of cloud applications," Ph.D. dissertation, The University of Waterloo, 2016.

[35] B. Suleiman, "Modeling and evaluation of rule-based elasticity for cloud-based applications," Ph.D. dissertation, The University of New South Wales, 2015.

[36] P. Sobeslavsky, "Elasticity in cloud computing," Master's thesis, Joseph Fourier University, 2011.

[37] [Online]. Available: https://aws.amazon.com/autoscaling

[38] [Online]. Available: https://aws.amazon.com/elasticbeanstalk

[39] [Online]. Available: https://aws.amazon.com/elasticloadbalancing

[40] [Online]. Available: http://www.fujii.nuee.nagoya-u.ac.jp/multiview-data/

[41] R. Skerjanc and J. Liu, "A three camera approach for calculating disparity and synthesizing intermediate pictures," *Signal Processing: Image Communication*, 1991.

[42] M. Ott, J. P. Lewis, and I. Cox, "Teleconferencing eye contract using a virtual camera." ACM, 1993.

[43] C. L. Zitnick, S. B. Kang, M. Uyttendaele, S. Winder, and R. Szeliski, "High-quality video view interpolation using a layered representation," in *ACM Transactions on Graphics (TOG)*. ACM, 2004.

[44] A.-R. Mansouri and J. Konrad, "Bayesian winner-take-all reconstruction of intermediate views from stereoscopic images," *IEEE Transactions on Image Processing*, 2000.

[45] M. Okutomi and T. Kanade, "A multiple-baseline stereo," *IEEE Transactions on pattern analysis and machine intelligence*, 1993.

[46] T. Kanade, P. Narayanan, and P. W. Rander, "Virtualized reality: Concepts and early results," in *RepresenProceedings IEEE Workshop on representation of Visual Scenes*, 1995.

[47] J.-I. Park, G. M. Um, C. Ahn, and C. Ahn, "Virtual control of optical axis of the 3dtv camera for reducing visual fatigue in stereoscopic 3dtv," *ETRI journal*, 2004.

[48] F. J. Halim and J. S. Jin, "View synthesis by image mapping and interpolation," in *Proceedings of the Pan-Sydney Area Workshop on Visual Information Processing - Volume 11*, 2001.

[49] H.-C. Shin, Y.-J. Kim, H. Park, and J.-I. Park, "Fast view synthesis using gpu for 3d display," *IEEE transactions on Consumer Electronics*, 2008.

[50] N. Fukushima, T. Fujii, Y. Ishibashi, T. Yendo, and M. Tanimoto, "Real-time free viewpoint image rendering by using fast multi-pass dynamic programming," in *3DTV-Conference: The True Vision-Capture, Transmission and Display of 3D Video (3DTV-CON)*. IEEE, 2010.

[51] Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, and D. Nister, "Real-time global stereo matching using hierarchical belief propagation," in *BMVC*, vol. 6, 2006.

[52] L. Wang, M. Liao, M. Gong, R. Yang, and D. Nister, "High-quality real-time stereo using adaptive cost aggregation and dynamic programming," in *Third IEEE International Symposium on 3D Data Processing, Visualization, and Transmission*, 2006.

[53] Y. Mori, N. Fukushima, T. Yendo, T. Fujii, and M. Tanimoto, "View generation with 3d warping using depth information for ftv," *Signal Processing: Image Communication*, 2009.

[54] L. Toni, G. Cheung, and P. Frossard, "In-network view synthesis for interactive multiview video systems," *IEEE Transactions on Multimedia*, 2016.

[55] B. Suleiman, S. Sakr, R. Jeffery, and A. Liu, "On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure," *Journal of Internet Services and Applications*, 2012.

[56] P. C. Brebner, "Is your cloud elastic enough?: performance modelling the elasticity of infrastructure as a service (iaas) cloud applications," in *Proceedings of the 3rd ACM/SPEC international Conference on Performance Engineering*. ACM, 2012.

[57] R. Ghosh, K. S. Trivedi, V. K. Naik, and D. S. Kim, "End-to-end performability analysis for infrastructure-as-a-service cloud: An interacting stochastic models approach," in *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*. IEEE, 2010.

[58] R. Ghosh, F. Longo, V. K. Naik, and K. S. Trivedi, "Modeling and performance analysis of large scale iaas clouds," *Future Generation Computer Systems*, 2013.

[59] P. C. Brebner, "Performance modeling for service oriented architectures," in *Companion of the 30th international conference on Software engineering*. ACM, 2008.

[60] A. Lenk, M. Menzel, J. Lipsky, S. Tai, and P. Offermann, "What are you paying for? performance benchmarking for infrastructure-as-a-service offerings," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011.

[61] J. Dejun, G. Pierre, and C.-H. Chi, "Ec2 performance analysis for resource provisioning of service-oriented applications," in *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*. Springer, 2010.

[62] S. Dutta, S. Gera, A. Verma, and B. Viswanathan, "Smartscale: Automatic application scaling in enterprise clouds," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012.

[63] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011.

[64] P. Marshall, K. Keahey, and T. Freeman, "site: Using clouds to elastically extend site resources," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE, 2010.

[65] J. Yang, J. Qiu, and Y. Li, "A profile-based approach to just-in-time scalability for cloud applications," in *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*. IEEE, 2009.

[66] W.-H. Liao, S.-C. Kuai, and Y.-R. Leau, "Auto-scaling strategy for amazon web services in cloud computing," in *IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, 2015.

[67] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014.

[68] J. Kwiatkowski, "Evaluation of parallel programs by measurement of its granularity." PPAM, 2001.

[69] P. Pacheco, *An introduction to parallel programming*. Elsevier, 2011.

[70] N. R. Herbst, S. Kounev, and R. H. Reussner, "Elasticity in cloud computing: What it is, and what it is not." ICAC, 2013.

Appendix A

View Synthesis Application Pipeline, Code Review

view synthesis()

This appendix presents selected data independent blocks of our view synthesis algorithm for parallel processing.

- *initialization and buffer allocation*
- *camera pose conversion*

  Compute left, right and virtual camera pose using intrinsics and extrinsics
- *homographies()*

  pre-compute homographies using camera Poses for all possible depth value [0-255]
- *calcVirtualDepth()*

  Using homographies and left/right depth
- *applyFilter()*

  Median and Bilateral filters to reject noise and fill holes
- *calcVirtualImage()*

  Using computed depth and left/right source Images

- *applyMorphology()*

  Combine virtual depths extracted from left and right views. Post process it to reduce artifacts.

- *cvAddWeighted()*

  Combine virtual images extracted from left and right views.

applyFilter()

```
void applyFilter(IplImage** in_out1, IplImage**  in_out2) {
int64_t time;
   static float total_t = 0;
    static int count = 1;
   time = cv::getTickCount();
   cvexMedian(*in_out1);
   cvexBilateral(*in_out1, 20, 50);
   cvexMedian(*in_out2);
   cvexBilateral(*in_out2, 20, 50);
   total_t +=(float)(cv::getTickCount() - time)/cv::getTickFrequency()
      *1000.0;
    timing_print("apply filter took %f msec, ave %f over %d frames\n",
              (float)(cv::getTickCount() - time)/cv::getTickFrequency()
                 *1000.0,
              total_t/count,
              count);
   count++;
}
```

calcVirtualDepth()

```
void* calcVirtDepth_thread(void* _tid) {
   int i, j;
   long tid = (long)_tid;
   info_print("Thread %ld starting...\n", tid);
   int local_h = (int)ceil(IMG_HEIGHT/NUM_THREADS + 1);
   int start_row = local_h * tid;
   int end_row = std::min(IMG_HEIGHT, (int)(tid+1)*local_h);
   uchar* udepth = (uchar*)udepth_left->imageData;
   uchar* udepth2 = (uchar*)udepth_right->imageData;
   CvMat* m = cvCreateMat(3, 1, CV_64F);
   CvMat* mv = cvCreateMat(3, 1, CV_64F);
   for(j = start_row; j < end_row; j++) {
       for(int i = 0; i < IMG_WIDTH; i++) {
           int pt = i + j * IMG_WIDTH;
           cvmSet(m, 0, 0, i);
           cvmSet(m, 1, 0, j);
           cvmSet(m, 2, 0, 1);
           uchar val = (uchar)depth_left->imageData[pt];
           cvmMul(homog_LV[val], m, mv);
           int u = mv->data.db[0] / mv->data.db[2];
           int v = mv->data.db[1] / mv->data.db[2];
           u = abs(u) % IMG_WIDTH; // boundary check
            v = abs(v) % IMG_HEIGHT;
           int ptv = u + v * IMG_WIDTH;
           udepth[ptv] = (udepth[ptv] > val) ? udepth[ptv] : val;
           val = (uchar)depth_right->imageData[pt];
           cvmMul(homog_RV[val], m, mv);
           u = mv->data.db[0] / mv->data.db[2];
           v = mv->data.db[1] / mv->data.db[2];
           u = abs(u) % IMG_WIDTH;
           v = abs(v) % IMG_HEIGHT;
```

```
            ptv = u + v * IMG_WIDTH;
            udepth2[ptv] = (udepth2[ptv] > val) ? udepth2[ptv] : val;
}
}
   cvReleaseMat(&m);
   cvReleaseMat(&mv);
   pthread_exit((void*) _tid);
}
```

calcVirtualImage()

```
void* calcVirtImage_thread(void* _tid) {
    long tid = (long)_tid;
    info_print("Thread %ld starting...\n", tid);
    int local_h = (int)ceil(IMG_HEIGHT/NUM_THREADS + 1);
    int start_row = local_h * tid;
    int end_row = std::min(IMG_HEIGHT, (int)(tid+1)*local_h);
    uchar* udepth = (uchar*)udepth_left->imageData;
    uchar* udepth2 = (uchar*)udepth_right->imageData;
    CvMat* m = cvCreateMat(3, 1, CV_64F);
    CvMat* mv = cvCreateMat(3, 1, CV_64F);
    for(int j = start_row; j < end_row; j++) {
       for(int i = 0; i < IMG_WIDTH; i++) {
       int ptv = i + j * IMG_WIDTH;
       mv->data.db[0] = i;
       mv->data.db[1] = j;
       mv->data.db[2] = 1;
       cvmMul(homog_VL[udepth[ptv]], mv, m);
       int u = m->data.db[0] / m->data.db[2];
       int v = m->data.db[1] / m->data.db[2];
```

```
        u = abs(u) % IMG_WIDTH;
        v = abs(v) % IMG_HEIGHT;
        int pt = u + v * IMG_WIDTH;
        for(int k = 0; k < 3; k++)
           dst_left_g->imageData[ptv * 3 + k] = src_left_g->imageData[pt * 3 +
               k];
        cvmMul(homog_VR[udepth2[ptv]], mv, m);
        u = m->data.db[0] / m->data.db[2];
        v = m->data.db[1] / m->data.db[2];
        u = abs(u) % IMG_WIDTH;
        v = abs(v) % IMG_HEIGHT;
        pt = u + v * IMG_WIDTH;
        for(int k = 0; k < 3; k++)
           dst_right_g->imageData[ptv * 3 + k] = src_right_g->imageData[pt * 3
               + k];
  }
  }
     cvReleaseMat(&m);
     cvReleaseMat(&mv);
     pthread_exit((void*) _tid);
  }
```

Appendix B

Main Web Service Application

This appendix represents main service application in Python deployed on AWS CloudWatch. It creates HTML form to get user's request and deliver a test message.

```
import subprocess as sp

# App config.
DEBUG = True
application = Flask(__name__)
application.config.from_object(__name__)
application.config['SECRET_KEY'] = '7d441f27d441f27567d441f2b6176a'

def create_load(load=0.9, duration=25, cpu=0):

    command = ["python", "./CPULoadGenerator.py", "-l", str(load), "-d",
        str(duration), "-c", "0"]
    command1 = ["./CPULoadGenerator.py", "-l", str(load), "-d", str(duration),
        "-c", "1"]
    command2 = ["./CPULoadGenerator.py", "-l", str(load), "-d", str(duration),
        "-c", "2"]
    command3 = ["./CPULoadGenerator.py", "-l", str(load), "-d", str(duration),
        "-c", "3"]
```

```
        #"|", "./CPULoadGenerator.py", "-l", str(load), "-d", str(duration),
            "-c", "1",
        #"|", "./CPULoadGenerator.py", "-l", str(load), "-d", str(duration),
            "-c", "2",
        #"|", "./CPULoadGenerator.py", "-l", str(load), "-d", str(duration),
            "-c", "3"]
    #print(command)

    proc = sp.Popen(command, stdout=sp.PIPE, stderr=sp.PIPE, bufsize=1)
    #proc = sp.Popen(command1, stdout=sp.PIPE, stderr=sp.PIPE, bufsize=1)
    #proc = sp.Popen(command2, stdout=sp.PIPE, stderr=sp.PIPE, bufsize=1)
    # proc = sp.Popen(command3, stdout=sp.PIPE, stderr=sp.PIPE, bufsize=1)
    #proc = sp.Popen(["python", "./CPULoadGenerator.py", "-l", str(load),
    #           "-d", str(duration), "-c", str(cpu)],
    #            stdout=sp.PIPE, stderr=sp.PIPE, bufsize=1)
    ll = []
    with proc.stdout:
        for line in proc.stdout:
          ll.append(line)
    with proc.stderr:
        for line in proc.stderr:
          ll.append(line)

    proc.wait()

    str_out = ""
    for l in ll:
        str_out += str(l) + " "
    str_out = "\n".join(str_out.split("\\n"))
    return str_out

header_text = '''
```

```
    <html>\n<head> <title>EB Flask Test</title> </head>\n<body>'''

home_link = '<p><a href="/">Back</a></p>\n'
footer_text = '</body>\n</html>'


class ReusableForm(Form):
    name = TextField('Name:', validators=[validators.required()])

@application.route("/favicon.ico", methods=['GET', 'POST'])
def favicon():
    return header_text

@application.route("/req", methods=['GET', 'POST'])
def req():

    print("I am created load")
    load_status = create_load(0.85, 5)

    return load_status

@application.route("/", methods=['GET', 'POST'])
def hello():
    form = ReusableForm(request.form)

    print(form.errors)
    if request.method == 'POST':
        name=request.form['name']
        print("I am created load")
        load_status = create_load(0.85, 5)

        if form.validate():
```

```
        flash(load_status + "\n" + name )
    else:
        flash('All the form fields are required. ')

    return render_template('hello.html', form=form)

if __name__ == "__main__":
    application.run()
```