Title
Large-Scale 3D Reconstruction on the GPU

Author
Mak, Jason

Publication Date
2022

Peer reviewed|Thesis/dissertation

Large-Scale 3D Reconstruction on the GPU

By

JASON MAK
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John D. Owens, Chair

Zhaojun Bai

Nina Amenta

Committee in Charge

2022

CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Code Listings

ABSTRACT

Large-Scale 3D Reconstruction on the GPU

3D multiple-view reconstruction is an important topic with applications in robotics, surveillance, augmented reality, and other fields. The ubiquity of reconstruction makes it a vital component in many systems, which need hardware and algorithms capable of processing the vast data found in reconstruction. In this dissertation, we first propose a method for performing a stage of reconstruction, triangulation, on the GPU that does not require second-order optimization and yields an order-of-magnitude speedup over a multi-core CPU processor. Next, we accelerate triangulation further on the GPU by discussing a method that leverages the path of a moving camera as a constraint and thus avoids doing non-linear optimization altogether. Subsequently, we shift to the problem of dense stereo, where we use GPU processing power to create more complete dense reconstructions while again leveraging scene constraints to keep runtime tractable. Finally, we devote a large portion of this dissertation to studying the use of multiple GPUs to accelerate the most expensive stage in reconstruction, bundle adjustment. The strategy is to partition the scene and optimize the subproblems in parallel. This approach minimizes communication across GPUs and removes the need for multi-GPU synchronization, which can be costly when there are many GPUs. We analyze multiple parallel, partitioned bundle adjustment strategies and their advantages and disadvantages. We develop an alternate method that parallelizes more efficiently and is scalable on large datasets with more GPUs. We compare the performance of the partitioned, parallel implementation against that of the original, full-problem implementation. We confirm our hypothesis that our approach can obtain large speedups with competitive accuracy for certain scenes but is less robust to ill-conditioned problems and the presence of local minima.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. John Owens, as well as my other committee members, Dr. Nina Amenta and Dr. Zhaojun Bai, for guiding me through research and providing me with helpful comments for my dissertation. I would also like to thank anyone with whom I have collaborated throughout my research. In particular, special thanks is given to my colleague, Dr. Mauricio Hess-Flores, for all his insightful help and advice. During my time in graduate school, my family has been supportive, for which I am grateful. Finally, I would like to thank the UC Davis College of Engineering for providing me with a stimulating academic environment to increase my knowledge and development in both teaching and research, and to the UC Davis Music Department and in particular, the UC Davis Video Game Orchestra, for enriching my studies with art and culture during my time in graduate school. Some of the work presented in this material was supported by the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. Any opinion, findings, and conclusions expressed in this work are those of the author(s) and do not necessarily reflect the views of the National Nuclear Security Administration or the U.S. government.

Chapter 1

Introduction

Multi-view 3D reconstruction is a computer vision problem that aims to reconstruct 3D objects or scenes using input images taken from cameras at different positions and angles. The principles are similar to the way humans perceive depth and 3D structure using visual information derived from two eyes. In the computer vision approach, 2D correspondences are located across different images and, along with the camera information, the 2D correspondences are used to solve 3D points. The output of reconstruction is often a point cloud, a set of 3D points that are consistent across images and cameras when projected back into 2D. They satisfy relationships based on the principles of trigonometry. Additional post-processing is often applied to a resulting point cloud to remove noise and to create a mesh from the set of points.

Multi-view 3D reconstruction has become popular in research and industry, with many useful applications including security, industrial design, virtual environment creation and enhancement, mobile robotics, and reconstruction of cultural heritage sites. Reconstruction is ubiquitous in scale ranging from large-scale aerial reconstruction, which focuses on images or video frames taken by aircraft of an underlying scene, to small-scale reconstruction done by mobile devices. Experts predict that unmanned aerial vehicles (UAV) will become common tools for government and commercial use in the future, and giving them awareness of the environment below allows for increased autonomy. On a different level, the mobile revolution has placed a computer into millions of hands, and users are increasingly using their devices to give them quick information about their world and surroundings. For example, Googles ARCore platform can reconstruct a users 3D environment to provide the users mobile device with a human-scale

understanding of space and motion from which other information can be derived.

Increasing the accuracy, efficiency, and density of reconstructions are areas of ongoing research in the community. VisualSfM [90, 91] is an open-source tool that uses bundle adjustment for error correction and Patch-based Multi-view Stereo (PMVS) [24] to generate dense point clouds. In addition, the popular open-source computer vision library, OpenCV, implements many routines used in the reconstruction process including feature detection, pose estimation, and bundle adjustment [11]. However, OpenCV is designed to be a general-purpose library used for a variety of computer vision tasks. It falls on the users to design their own applications, such as reconstruction, using the provided routines, with no awareness from the library of how data flows through the application.

Many challenges exist in reconstruction, often resulting from the limitations in available data or a lack of data. Poor lighting in images, shadows, and obstructed views can degrade the techniques used in the correspondence problem. Certain properties of physical cameras, such as distortion, may be unknown or hard to model, which causes issues. Some of these concerns may become less of an issue over time as the size and availability of collected input data continues to increase. Cameras are easily accessible with ever-increasing pixel density. Consumer smartphones are equipped with cameras that can capture pictures of many megapixels. Improved storage systems have allowed for the immense collection of such pictures, and advanced communication technologies make them widely accessible across vast distances. Furthermore, the increased availability and efficiency of various types of sensors yield new types of input data that add to the size of the reconstruction working set. Such data includes LIDAR scans and GPS coordinates, which can help augment or increase the accuracy of reconstructions generated using traditional multi-view stereo methods [5].

With improved data collection, the scale of many reconstruction problems has grown due to the types of scenes that are increasingly common to model, such as those of terrain and cities. Since the time photography was invented, photogrammetry has been used, often manually, to create topographic maps [56]. The mathematical principles are still valid today. More recently, the advent of UAVs, both government and consumer, enable the bulk capture of aerial footage at a large scale. Reconstructions derived from these images have increasingly diverse applica-

tions, such as reconnaissance, traffic monitoring, agriculture, and the creation of realistic virtual worlds for entertainment purposes. Reconstruction from aerial footage offers some advantages, since the aircraft may also be equipped with additional sensors, such as GPS and IMU, that can aid in the accuracy of reconstruction. With the popularity of drones and mobile devices, onboard processing is also becoming desirable, as wireless communication for passing data can potentially be a huge bottleneck.

Software has been developed that target these types of large-scale scenes and applications. Google Earth reconstructs cities in 3D by leveraging large-scale data processing algorithms [36]. The target development platforms include large computing clusters that are not readily available to the average user. Pix4D can reconstruct scenes given an arbitrary set of input images. Their algorithms work well with aerial photography and can generate highly realistic reconstructions [80]. SocetGXP, a photogrammetry software developed by BAE Systems, is used by the military and other organizations to perform reconstructions using aerial and satellite imagery, as well as other supporting data such as terrain maps. Although these software tools can potentially produce high-quality results, the methods they use are often proprietary.

The increasing scale of reconstruction problems has also spurred research into the creation of large benchmarks with ground truth that can be used to evaluate different software and methods. Strecha et al. [82] created the EPFL benchmark, which features outdoor building facades and uses LiDAR scans for ground truth. The dataset of Merrell et al. is of a single building, but the input is large-scale in that it consists of a dense collection of video frames [54]. In more recent works, Schps et al. developed a new benchmark for two and multi-view stereo algorithms [75]. Knapitsch et al. developed benchmarks of both indoor and outdoor environments to test full reconstruction pipelines [40]. They use a laser scanner to create ground-truth reference models. A common goal of the last two benchmarks is to not only provide ground truth for measuring accuracy, but also to be future-proof by using high-resolution cameras (large images) and video data (numerous images). This ensures that the benchmarks will continue to remain representative of modern, data-intensive Structure-from-Motion problems.

As a trade-off, improved data collection capabilities and larger-scale problems leads to an increase in runtime. With performance becoming a greater issue, the advent of GPU computing

has not been ignored in the 3D reconstruction community. GPUs were traditionally designed to target computer graphics workloads, but have since evolved to perform general-purpose computing, including scientific simulations, linear algebra, and computer vision. Many operations in reconstruction are highly parallel, involving image-processing tasks and workloads similar to those used in computer-graphics, GPUs are well-equipped to process a large amount of streaming data with their high memory bandwidth and numerous cores. Many of the stages of reconstruction have been accelerated on the GPU. These include the mapping of existing algorithms and the development of new ones, as will be discussed later.

The contributions of this dissertation focus on increasing the performance of 3D reconstruction stages through GPU parallelization while maintaining or exceeding the accuracy of existing methods. Some ideas are designed to exploit the domain-specific structure of 3D reconstruction problems. The following is a layout of our contributions.

- In Chapter 3, we implement large-scale triangulation by using the angles between camera-point rays to produce a massively parallelizable implementation and avoid second-order optimization.

- In Chapter 4, we use the path of a single moving camera in a scene as a strong constraint that allows us to avoid doing non-linear optimization altogether and is easily parallelizable.

- In Chapter 5, we use the assumption that depth values for each object in an image can vary smoothly. This allows us to pick a good starting point when doing a brute-force search to find the correct depth of each pixel.

- In Chapter 6, we do large-scale bundle adjustment by taking advantage of the fact that reconstruction problems can be partitioned spatially. Using this approach to parallelize the large non-linear optimization problem minimizes the communication needed across multiple GPUs. We release the parallel bundle adjustment source code on Github.

In the next chapter, we introduce a basic 3D reconstruction pipeline and provide an overview of each stage.

Chapter 2

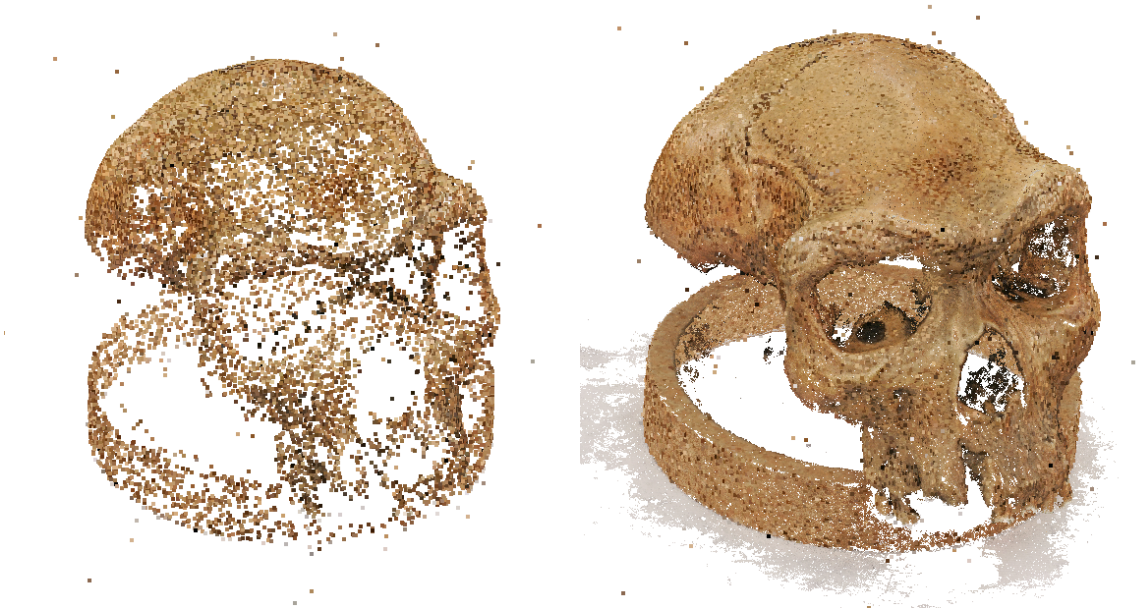
Background

In this chapter, we first give a brief introduction to the reconstruction problem. In the following sections, we provide an overview of each stage in a basic reconstruction pipeline. The layout of this chapter is arranged as follows.

1. Introduction [Section 2.0]
2. Camera Calibration [Section 2.1]
3. Feature Detection and Matching [Section 2.2]
4. Pose Estimation [Section 2.3]
5. Triangulation [Section 2.4]
6. Bundle Adjustment [Section 2.5]
7. Dense Stereo [Section 2.6]
8. GPUs and Reconstruction [Section 2.7]

2.0 Introduction

There are many variations of the reconstruction pipeline. Reconstruction across the literature usually involves a common base set of stages that includes feature tracking, self-calibration, camera pose estimation, structure computation, and parameter optimization [30]. First, camera



(a) sparse (b) dense

Figure 2.1: Example 3D reconstructions of the *skull* dataset [24].

calibration is used to determine a cameras intrinsic parameters, such as the focal length and the principal point. Next, feature detection is performed, using popular methods such as SIFT [51] and SURF [6]. Mappings between corresponding features in different images of the scene are computed. The next step is to compute the epipolar geometry, which can be estimated from the matches in a feature track. Pose estimation is then used to obtain the cameras' extrinsic parameters: translation and rotation. With the obtained camera parameters and a set of chosen feature tracks, triangulation is used to compute the 3D structure. There is the possibility for inaccuracies to occur in the steps leading to the computation of the 3D structure. A further step, bundle adjustment, can be used to solve an optimization problem, ensuring that the different parameters of the solution are as consistent with each other as possible. In the next sections, we provide a brief overview of each of these steps. They are more formally defined in the well-known book by Hartley and Zisserman [30]. The final section also discusses the use of GPUs in reconstruction. Example reconstruction results from the *skull* dataset [24] are shown in Figure 2.1.

2.1 Camera Calibration

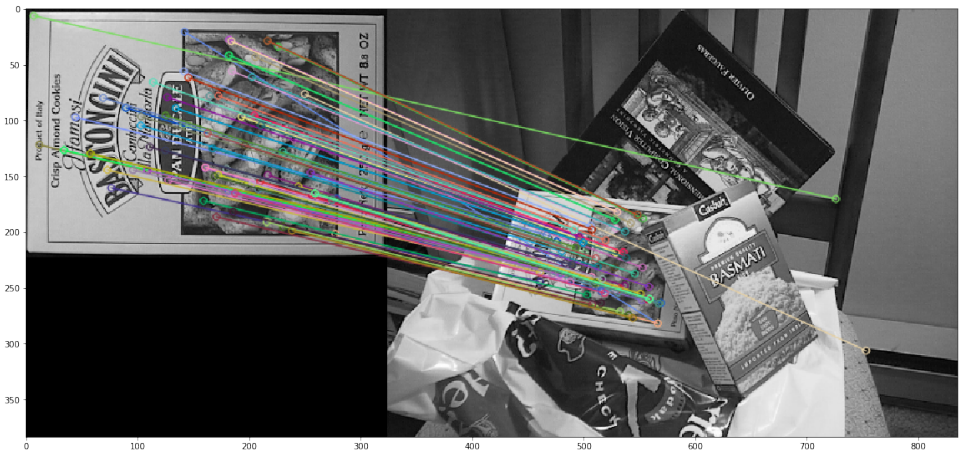
To begin reconstruction, we need to perform camera calibration, also known as camera resectioning. The goal is to retrieve the relevant intrinsic parameters of the cameras used to take the images. Common intrinsic parameters include focal length, principal point, and pixel skew. These are fundamental properties of the camera determined by the manufacturers, which affect how a scene is mapped onto a 2D plane, forming an image. Often, this information can be found in specifications that come with the camera. In addition, they may be stored in the metadata of an image, i.e. through EXIF tags in the taken images. If the data is not stored with images and the source cameras for the images are unknown, then we must perform manual calibration. We must use a camera model that encapsulates all the necessary parameters, and then through feature detection and matched features, we can solve for the unknown parameters. The typical model for the intrinsic properties of a camera is a 3x3 matrix often labeled as K, as shown in Equation 2.1. Here, f is the focal length, fk_u is the scaling in the x-direction, fk_v is the scaling in the y-direction, and (u_0, v_0) is the principal point.

$$K = \begin{bmatrix} fk_u & 0 & u_0 \\ 0 & -fk_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

2.2 Feature Detection and Matching

The goal of feature detection is to find and label points of interest, or keypoints, in the input images. Corners or bright spots that stand out in the images are often points that can be used as features. Figure 2.2 shows an example of detected features in images of a food package. Each keypoint, along with other information, such as surrounding pixels, make up a feature. The same feature and its different locations in multiple images make up a feature track, as we are tracking the feature throughout these images.

There are both sparse and dense methods for feature representation and detection. Popular sparse methods use neighborhood gradient magnitude and direction information to represent a feature. SIFT is one of the more robust ways to represent and detect features. SIFT first finds keypoints in the image by finding maxima and minima of the differences of Gaussian functions



(a)

Figure 2.2: Example of feature matching. The colored lines connect matched pixels corresponding to the same scene point in two images of two different views of a food package.

applied to smoothed and resampled images. For each keypoint, SIFT creates a 128D vector based on local gradient information, and this, combined with the keypoint's 2D location, is used to distinguish a feature. For each feature in a reference image, finding the corresponding feature in other images involves matching features that have the minimal Euclidean distance between their 128D vectors. A feature matched across multiple images creates a feature track. Figure 2.2 shows a feature track with matched features across different images. In addition to sparse methods, there are dense feature matching algorithms such as those based on optical flow, which try to exploit motion cues [6]. While the contributions in this dissertation do not utilize them, an overview of dense feature methods can be found in the work of Scharstein [74].

Creating accurate feature tracks can be a challenge due to ambiguities caused by occlusions, shadows, specular highlights, and repeated textures. A good reconstruction pipeline needs to account for such inaccuracies and seek to correct them. In Chapter 4, we introduce a method that uses the path of a moving camera as a constraint to produce a 3D structure, while simultaneously correcting inaccurate feature tracks. We also map this highly parallelizable method to the GPU to obtain faster results than previous methods.

2.3 Pose Estimation

In addition to the intrinsic parameters for a camera described previously, one also needs the extrinsic parameters, such as its position and rotation. These data values may be provided by other sensors, such as IMU and GPS, and certain reconstruction apparatuses, like those in aerial reconstruction and modern mobile computer vision platforms, are more likely to include such additional sensor data, which augment the images taken by cameras. Traditionally, for the most generalized reconstruction applications, these camera values are not available and need to be calculated relative to each other post-image collection. This is done via pose estimation. To perform pose estimation, we first have to discuss the concept of epipolar geometry. When considering two different images of the same scene, the epipolar geometry is the intrinsic projective geometry between the two views and is encapsulated in a 3x3 matrix called the fundamental matrix. In the previous section, we discussed the 3x3 camera matrix. The fundamental matrix can be computed from the camera matrices in the left and right images (K and K') and another matrix E, the essential matrix, as shown in Equation 2.2. The essential matrix holds information about the relative rotation and translation between a pair of cameras. How do we find the fundamental matrix? Feature matches in two images enables us to to solve for the fundamental matrix between two cameras. Without delving into much detail, the 2D positions of 8 feature matches can be used to create a linear system, where the unknowns are the entries of the fundamental matrix. Fewer than 8 feature matches can be used as well, but this may require solving a non-linear system.

$$F = K^{-T} E K'^{-1} \tag{2.2}$$

Once we have the matrices K and K' from the intrinsic camera parameters and the fundamental matrix F, it is easy to see that the essential matrix E can be derived. Afterwards, the E matrix can be decomposed into a relative 3x3 rotation matrix R and a relative 3x1 translation vector T using Singular Value Decomposition. The process can lead to four possible solutions of (R, T). We choose the correct solution through a simple test that ensures depths of the scene points are positive in both cameras. The first camera is typically used as the reference, with its position placed at the origin and its orientation set at zero rotation. R and T for other cameras

are solved with respect to the reference camera.

With the intrinsics and extrinsics for a given camera, we can now create a final matrix for this camera called a camera projection matrix. Here, we keep the rotation matrix R and translation vector T as separate entities within a single matrix. Then, we multiply the camera intrinsic matrix by this matrix to get a camera projection matrix P, shown in Equation 2.3. This matrix describes the mapping of a 3D point X in the scene to a 2D point x in the image produced by that camera, as shown in Equation 2.4. The camera projection matrix is a data format for cameras frequently used in the literature and the one we use for work throughout this proposal. More details on the fundamentals of epipolar geometry and the camera projection matrix can be found in the book of Hartley and Zisserman [30].

$$P = K \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \tag{2.3}$$

$$x = P \begin{bmatrix} X \\ 1 \end{bmatrix} \tag{2.4}$$

2.4 Triangulation

Triangulation computes the 3D structure of a scene using the projection matrices and the 2D information contained in the feature tracks. For a given feature track, the 2D positions of the features in the track and the projection matrices can be used to solve for the 3D position of the scene point in world space, To visualize this, assume that a feature track and the projection matrices are perfectly correct. For each image in the track, if we shoot a ray from each image's camera center through the pixel corresponding to the feature, the rays for all cameras should intersect perfectly at one point. In practice, due to error in various parts of the process, the rays will not perfectly intersect and a best-fit point must be found based on minimizing some sort of error metric.

A more detailed review of triangulation and its various implementations will be discussed later. Triangulation is an important step in reconstruction as it produces a 3D scene structure for the first time. Furthermore, the result of triangulation acts as a starting point for the next stage, bundle adjustment, in which the feasibility of a non-linear optimization depends greatly on the

starting point. Triangulation can be done multiple times throughout the reconstruction process to help maintain accuracy. Therefore, reconstruction pipelines aim to use a triangulator that is both accurate and fast. In Chapter 3, we introduce a GPU-based triangulator that maintains or exceeds the accuracy of common methods. Our triangulator uses an easily parallelizable L_1 cost function and obtains large speedups.

2.5 Bundle Adjustment

A resulting reconstruction might have errors accumulated from multiple stages. In the absence of ground truth data, reprojection error is the only valid metric to assess the accuracy of a reconstruction. Reprojection error is the sum of squares of the Euclidean distances between a computed 3D point's 2D projection and its corresponding feature track locations across all images in which the point is visible. A common final stage in reconstruction is bundle adjustment, which is an optimization of these parameters, such that the reprojection error is minimized. The minimization is achieved using non-linear least-squares algorithms, among which Levenberg-Marquardt has proven to be one of the most successful. Harley and Zisserman provides further information on bundle adjustment [30].

It is difficult to analyze reprojection error to determine the cause of such error. Total reprojection error does not provide any information regarding the sources of the error. For instance, a small number of feature tracks with high reprojection error could be the main component of total reprojection error, and bundle adjustment optimization, despite its expensive computation, may not be able to lower its value. Bundle adjustment performs a non-linear optimization of the sum-of-squares reprojection error function, which is prone to miring in local minima and may not converge to the global optimal solution. If inaccurate feature tracks could be removed or corrected, total reprojection error may be much lower, and bundle adjustment would converge much faster with a greater probability of obtaining the global minimum.

For large-scale problems, the non-linear optimization required by bundle adjustment can pose a challenge, especially since bundle adjustment can be done multiple times periodically throughout the reconstruction process. Ongoing research seeks to use parallel computing to reduce the runtime of bundle adjustment. In Chapter 6, we present a divide-and-conquer approach

to the bundle adjustment problem, where we partition the problem into subproblems separated by boundary points. We alternate optimizing the subproblem parameters with the boundaries fixed and optimizing the boundaries with the subproblem parameters fixed. The method takes advantage of a multi-GPU system to gain large speedups.

2.6 Dense Stereo

Another stage of reconstruction, dense stereo, produces a dense reconstruction and is optional depending on the application. Figure 2.1 shows an example of a sparse and dense reconstruction of the same object. For some applications, such as Simultaneous Localization and Mapping (SLAM), a sparse reconstruction can be adequate. In other cases, such as computer graphics and augmented reality, it is desirable to turn a sparse reconstruction into a realistic 3D object or scene. For dense stereo, the accuracy of the cameras are highly important and are largely determined by the outcomes of the previous stages. With accurate enough cameras, the pixels in each image can be used to "fill in the gaps" in the sparse reconstruction to produce a dense, realistic 3D object. Some strategies use geometric meshing to create a dense reconstruction, while others might simply add more points to the sparse point cloud to create a dense point cloud. Since it is desirable to leverage as much image data as possible, dense stereo can be computationally expensive due to the amount of images available in a large-scale reconstruction dataset. In Chapter 5, we introduce a method that leverages 2D scene information found in the images along with multiple GPUs to produce highly dense point clouds within a reasonable time.

2.7 GPUs and Reconstruction

As mentioned previously, GPUs are useful tools for accelerating a variety of applications. The most popular GPU computing platform, with a unique architecture and its corresponding programming model, is NVIDIA's Compute Unified Device Architecture (CUDA). We will be using this programming model throughout this proposal. CUDA is well-known today, so we will only explain it briefly here and defer to Nickolls et al. [60] to describe the model in more detail. CUDA programs are called *kernels*, which are run on a collection of parallel *blocks*, each of which contains up to 1024 *threads*. The GPU assigns blocks of threads to one of its

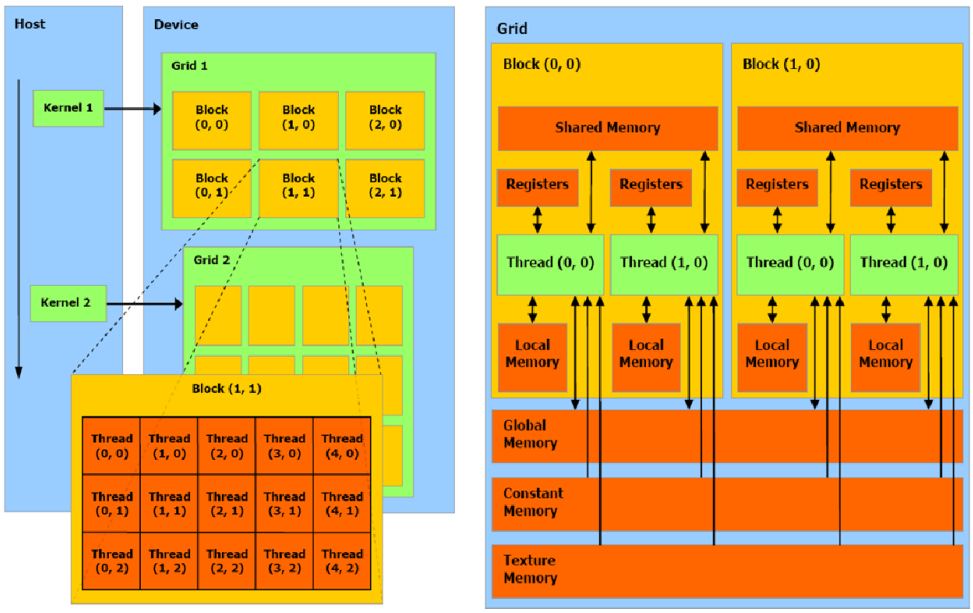


Figure 2.3: The CUDA host-device-kernel relationship and grid-block hierarchy is shown on the left. The block-thread hierarchy and memory hierarchy is shown on the right. These images are taken from the CUDA Programming Guide [63].

many *streaming multiprocessors* (SMs), which runs groups of 32 threads called *warps* in lock-step under SIMD control. Threads within a block can also share data through a small *shared memory* that is over 100 times faster than off-chip DRAM (*global memory*). Another type of memory, texture memory, is potentially faster than global memory if the memory accesses are irregular but have 2D spatial locality. Efficient GPU programs must fill the machine with work by launching a large number of threads; must minimize thread divergence (threads that take different control flows) within warps; and must efficiently use the memory hierarchy, using fast shared memory in preference to global memory if at all possible. Throughout the evolution of the CUDA, new features have been added to give developers more flexibility. Support for global atomic operations on various data types enable programmers to synchronize computation globally within a kernel. Introduced in CUDA 9, cooperative groups allow a programmer-specified number of threads in a thread block up to the size of the block to communicate and synchronize with each other. Previously, such granularity of control existed only within a warp or across the entire block.

Many stages of reconstruction, especially those that are computationally expensive, have been implemented on the GPU. For feature detection, Wu developed the first GPU implemen-

tation of SIFT in 2007 [89]. The basics of the implementation are straightforward, as SIFT is highly parallel, since blurring images and taking image differences are essentially image processing tasks. Bjorkman et al. created a more recent implementation of SIFT on the GPU optimized for newer NVIDIA GPUs [9]. Bundle adjustment is another computationally-expensive stage in reconstruction, and non-linear solvers, particularly Levenberg-Marquardt (LM), have been implemented on the GPU. The most expensive step in LM is solving a linear system per iteration for multiple iterations. An iterative method, preconditioned conjugate gradient (PCG), is often used as the solver. Wu et al. and Zheng et al. both implement this method on the GPU for bundle adjustment, as the algorithm maps well to a highly parallel architecture [91, 97]. The most expensive step in PCG is sparse matrix-vector multiply, which is straightforward on the GPU. Other steps also involve basic linear algebra routines.

Other reconstruction applications, particularly those that target aerial and large-scale scenes, have been implemented on the GPU. Zhu et al. designed a end-to-end pipeline to go from aerial images to disparity images in real-time using the GPU [98]. Their workflow is simple and regular, and due to their domain of reconstruction based on aerial imagery, they have GPS coordinates to help with calibration. They perform a brute-force search for each pixel to find the best depth value. Yang et al. and Hane et al. used variations of the plane-sweeping stereo method to implement dense stereo reconstruction on the GPU [34, 93]. Plane-sweeping stereo discretizes the 3D scene into a stack of planes, providing a straightforward way to assign work among different parallel threads. OpenCV provides GPU backends for some of its routines, mainly those that involve image-processing algorithms. The transfers of data between host and device at the beginning and end of the routines are hidden to the user but may incur overhead. VisualSfM also provides the option of using the GPU for certain stages of reconstruction. Like OpenCV, data transfers must occur at the beginning and end of a stage.

In this dissertation, we focus on leveraging GPUs to accelerate the triangulation stage, bundle adjustment stage, and dense stereo stage of 3D reconstruction. In the following chapters, we cover our methods and their GPU implementation details. The next chapter introduces a GPU-based triangulator that is both fast and accurate.

Chapter 3

Large-Scale Triangulation on the GPU

In this chapter, we present a GPU N-view triangulator that maintains or exceeds the accuracy of common existing methods. Our method uses a parallelizable L_1 cost function and achieves up to 39x speedup over a serial implementation. Section 3.0 gives a definition of the triangulation problem, while Section 3.1 gives an overview of previous work on triangulation. Section 3.2 provides the details of our implementation, including the definition of the cost function, a analysis of its convexity, the use of statistical sampling, and the granularity of our parallelization on the GPU. Section 3.3 presents the results of running our implementation on synthetic and real data.

3.0 Definition of Triangulation in the Context of Computer Vision

Multi-view scene reconstruction involves a number of stages applied sequentially, where the output of one stage directly affects accuracy in the following steps. Triangulation is a key step in reconstruction. Its accuracy is a direct function of previously-computed feature tracking, camera intrinsic calibration, and pose estimation [30]. Typically, 3×4 projection matrices are used to encapsulate all camera intrinsic and pose information. Triangulation aims to determine the 3D location of a scene point, X, from its imaged pixel location, x_i, in two or more images. When X reprojects exactly onto its x_i coordinates, such that epipolar constraints [30] on the matches are perfectly satisfied, triangulation is trivial through even the simplest methods. However, in the presence of image noise, the reprojected coordinates of X will not coincide with each

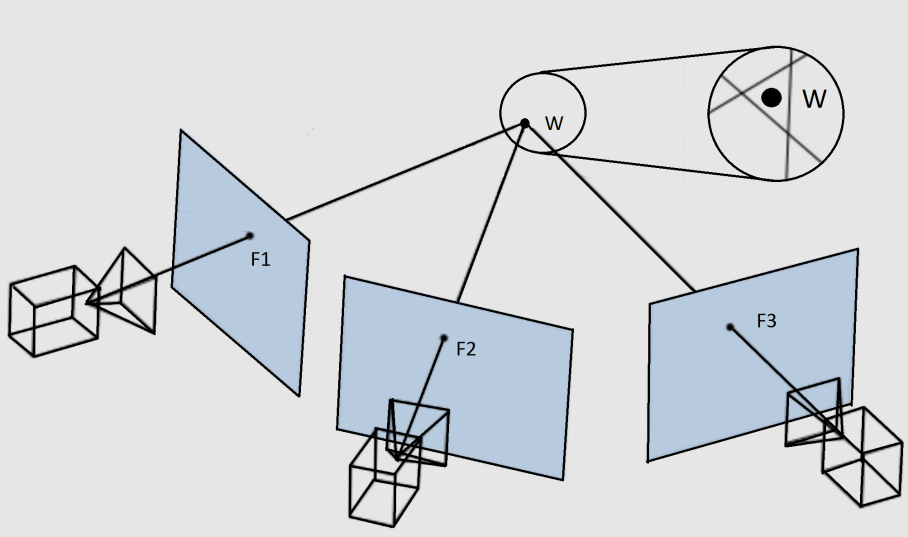


Figure 3.1: Across three images, a set of rays originating from the camera positions are passed through 2D features $F1$, $F2$, and $F3$ in an attempt to recover the 3D world point W. When zooming in on the rays' closest point of intersection, we will often see that there is no perfect solution. Triangulation is used to estimate the most likely position of the 3D world point.

respective x_i. In settings with an arbitrary number of cameras, noisy camera parameters, and inexact image measurements (*feature tracks*), the goal becomes finding the point X that best fits a given track.

The main issue with all previous triangulation algorithms is scalability. Improved data collection capabilities are increasing both the quality and quantity of data used as input for triangulation. Advancements in camera technology produce high-resolution images and the mobile revolution coupled with improved data storage and sharing techniques enable numerous users to generate images of the same scene. As the image resolution, number of images, and number of features within images grow, the process of triangulation can become intractable with current methods. Such issues arise for example in aerial reconstruction from UAVs. In addition, the ability to perform real-time processing is becoming desirable. Addressing issues of performance requires embracing modern tools for high-performance computing in software and hardware.

3.1 Related work

To this end, linear triangulation [30] is a fast method that is most widely used in the literature. The method solves for 3D points based on linear least squares, but given noisy inputs, the final result can be very inaccurate. A system of the form $AX = 0$ is solved by eigen-analysis or Singular Value Decomposition, where the A matrix is a function of feature track and camera projection matrix values. The obtained solution is a direct, best-fit solve, regardless of how noisy the inputs are. The solution is not optimal in that it doesn't minimize the L_2-norm of reprojection error. Numerical stability issues are also possible, especially with near-parallel cameras. The midpoint method [30] is by far the fastest method given two views, but it is very inaccurate in general. More recent methods have focused on achieving higher accuracy, typically by minimizing the L_2-norm of reprojection error through non-convex constrained optimization to achieve an *optimal* solution. This norm corresponds to the maximum-likelihood estimate for the point assuming independent Gaussian image noise.

There exist a number of optimal triangulation algorithms in the literature. One class of algorithms is based on *polynomial methods* [31], where all stationary points of a cost function are computed and evaluated to find the global minimum. The cost function must be expressed as a rational polynomial function in the parameters. The function's extrema are located where the derivatives with respect to the parameters become zero. The degree of the polynomial grows cubically with the number of views [81]. This implies a cubic growth in the number of local minima to evaluate, so this procedure has only been feasible for two and three views so far. Hartley and Sturm's optimal two-view method [29] applies an epipolar geometry-based Sampson correction on feature match positions x and x' to correct them such that they lie at the closest positions to epipolar lines. The correction requires solving for the stationary points of a 6th-order polynomial and then evaluating each real root. Lindstrom's "fast triangulation" algorithm [49] expresses the same set of equations in terms of Kronecker products, which by allowing terms to cancel out reduces the function to a quadratic equation and results in a one-to-four order-of-magnitude performance increase. Polynomial methods for three-views differ from two-view methods in that feature track positions are left intact. Kanatani et al. [38] develop a triangulator that outperforms Hartley and Sturm's method by avoiding singularities at the epipole

and using an iterative approach to get much faster performance. Stewénius et al. [81] applied the Gröbner basis method for solving polynomial equation systems. The real solutions for 47×47 action matrices are evaluated, where up to 24 minima may exist. Arithmetic operations are performed with 128 bits of mantissa to avoid round-off error accumulation. The method by Byröd et al. [12] alleviates such numerical issues by using the *relaxed ideal* modification for Gröbner bases, but at the expense of an even greater processing time.

A second class of algorithms is based on optimizing a cost function without seeking a direct solution like the polynomial-based algorithms. Some of these methods support N views. In general, these methods are promising but lack solid experimental results as far as error and processing time against different noise and camera configurations. Agarwal et al. [2] use fractional programming and a *branch and bound* algorithm to find a position arbitrarily close to the global optimum. There are also a few methods based on convex optimization on an L_∞ cost function. Hartley and Kahl [31] as well as Min [57] perform convex optimization on an L_∞ cost function making use of second-order cone programming (SOCP). Dai et al. [17] describe an L_∞ optimization method based on gradually contracting a region of convexity towards computing the optimum. In general, it is not clear how algorithms based on L_∞ behave under noise and for arbitrary numbers of cameras. It is also not justified why L_∞ was chosen over L_1, which behaves very well in Dai et al. [17].

N-view triangulation has traditionally been treated in two phases, where an initial linear method such as N-view linear triangulation [30] is applied to obtain an initial point followed by non-linear *bundle adjustment* optimization to reduce the sum-of-squares L_2-norm of reprojection error [50]. This procedure is prone to local minima, so a very accurate initialization is required.

A triangulator presented by Recker et al. [67] optimizes a L_1-based error cost function derived from the angles between camera rays. The method obtains an initial position through the midpoint method and applies adaptive gradient descent [79] on the angular cost function. This function is smoothly varying in a large basin in the vicinity of the global optimum, making it more robust to outliers and local minima than the L_2 norm of reprojection error. Furthermore, a statistical sampling component is introduced to increase efficiency without sacrificing accuracy.

A significant speed increase and better reprojection errors were obtained than with other triangulators, including N-view linear triangulation. However, the results are not provably optimal, and rely on a possibly inaccurate midpoint-based initialization.

There have also been triangulation algorithms developed for GPUs. Sánchez et al. [71, 72] present a GPU triangulator based on Monte Carlo simulations. Compared against Levenberg-Marquardt, they achieve the same precision but in much less time. However, the authors neither test their implementation on large-scale images with many features nor analyze how noise affects their results. Most reconstruction pipelines still use triangulators that run on CPUs. A more comprehensive overview and comparison of various triangulation methods is given in Strecha et al. [82].

3.2 Methodology

We introduce a fast and accurate GPU N-view triangulator. Our method is based on the cost function of Recker et al. [67], which is ideal for parallelization because it consists of abundant independent work. The following are the main contributions of our work:

- Using the CUDA programming model, we develop and compare two parallelization approaches for the GPU: using one *thread* to process one track and using one *thread block* to process one track. We show that the better-performing approach for a given dataset depends on the number of tracks and track lengths.

- Our algorithm uses statistical sampling based on confidence levels to successfully reduce the quantity of feature track positions needed to triangulate an entire track. We discuss how this sampling aids in our parallel implementation by improving load balancing and exploiting the GPU memory hierarchy.

- Finally, we test our GPU implementation on synthetic and real data. Our runtimes are up to 39x and 9x faster than contemporary serial and multi-core CPU implementations respectively, with final reprojection errors that are comparable to existing triangulators. This opens the door to triangulating large data very accurately and efficiently, a combination yet unseen in the triangulation literature.

3.2.1 Triangulation cost function

There are a number of cost functions in the vision literature. The L_2 least-squares solution is the maximum likelihood (ML) estimate under Gaussian image noise, but typically contains many local minima. The L_∞ model assumes uniform bounded noise and commonly results in a single solution. However, the L_1 norm measures the median of noise and is more robust to outliers than L_2 or L_∞, with desirable convergence properties.

Recker et al. [67] proposed an L_1 triangulation cost function, shown in Eq. 3.1, based on an angular error measure. Given C_i cameras, T the set of all feature tracks, and a 3D evaluation position $p = (X, Y, Z)$, the cost function for p with respect to a track $t \in T$ can be defined.

$$f_{t \in T}(p) = \frac{\sum_{i \in I}(1 - \hat{v_i} \cdot \hat{w_{ti}})}{||I||} \tag{3.1}$$

The inputs are a set of feature tracks across N images and their respective 3×4 camera projection matrices P_i. The method to compute the error for p is shown in Fig. 3.2(a) and explained as follows. A unit direction vector v_i is first computed between each camera center C_i and p. A second unit vector, w_{ti}, is computed as the ray from each C_i through its 2D feature track t in each image plane. Since t generally does not coincide with the projection of p in each image plane, there is frequently a non-zero angle between each possible v_i and w_{ti}. Finally, the average of the dot products $v_i \cdot w_{ti}$ across all cameras is obtained. Each dot product can vary from $[-1, 1]$, but only points that lie in front of the cameras are taken into account, corresponding to the range $[0, 1]$.

In Eq. 3.1, $I = \{C_i | t$ "appears in" $C_i\}$, $\vec{v_i} = (p - C_i)$, and $\vec{w_{ti}} = P_i^+ t_i$. The right pseudo-inverse of P_i is given by P_i^+, and t_i is the homogeneous coordinate of track t in camera i. The equation can be expanded with $v_i = (v_{i,X}, v_{i,Y}, v_{i,Z}) = (X - C_{i,X}, Y - C_{i,Y}, Z - C_{i,Z})$, with normalized $\hat{v_i} = \frac{v_i}{||v_i||}$ and $\hat{w_{ti}} = \frac{w_{ti}}{||w_{ti}||}$. In the absence of noise, Eq. 3.1 evaluates to zero at its minimum. However, the function is not convex and therefore must be solved iteratively as a non-linear optimization problem. The following defines the gradients along the X, Y, and Z directions [67], where each gradient is defined by an expression a, b, or c, and d is a common

denominator in all expressions.

$$\nabla f_{t \in T} = (\frac{\sum_{i \in I} a}{||I||}, \frac{\sum_{i \in I} b}{||I||}, \frac{\sum_{i \in I} c}{||I||}) \tag{3.2}$$

$$\begin{aligned} a = (&-C_{i,Y}^2 w_{\hat{t}i,X} - C_{i,Z}^2 w_{\hat{t}i,X} - C_{i,X} w_{\hat{t}i,Y} Y + w_{\hat{t}i,Y} XY - w_{\hat{t}i,X} Y^2 + \\ &C_{i,Y}(C_{i,X} w_{\hat{t}i,Y} - w_{\hat{t}i,Y} X + 2 w_{\hat{t}i,X} Y) - C_{i,X} w_{\hat{t}i,Z} Z + w_{\hat{t}i,Z} XZ - \\ &w_{\hat{t}i,X} Z^2 + C_{i,Z}(C_{i,X} w_{\hat{t}i,Z} - w_{\hat{t}i,Z} X + 2 w_{\hat{t}i,X} Z))/d \end{aligned} \tag{3.3}$$

$$\begin{aligned} b = (&-C_{i,X}^2 w_{\hat{t}i,Y} - C_{i,Z}^2 w_{\hat{t}i,Y} - C_{i,Y} w_{\hat{t}i,X} X - w_{\hat{t}i,Y} X^2 + w_{\hat{t}i,X} XY + \\ &C_{i,X}(C_{i,Y} w_{\hat{t}i,X} + 2 w_{\hat{t}i,Y} X - w_{\hat{t}i,X} Y) - C_{i,Y} w_{\hat{t}i,Z} Z + w_{\hat{t}i,Z} YZ - \\ &w_{\hat{t}i,Y} Z^2 + C_{i,Z}(C_{i,Y} w_{\hat{t}i,Z} - w_{\hat{t}i,Z} Y + 2 w_{\hat{t}i,Y} Z))/d \end{aligned} \tag{3.4}$$

$$\begin{aligned} c = (&-C_{i,X}^2 w_{\hat{t}i,Z} - C_{i,Y}^2 w_{\hat{t}i,Z} - C_{i,Z} w_{\hat{t}i,X} X - w_{\hat{t}i,Z} X^2 - C_{i,z} w_{\hat{t}i,Y} Y) - \\ &w_{\hat{t}i,Z} Y^2 + w_{\hat{t}i,X} XZ + w_{\hat{t}i,Y} YZ + C_{i,X}(C_{i,Z} w_{\hat{t}i,X} + 2 w_{\hat{t}i,Z} X - \\ &w_{\hat{t}i,X} Z) + C_{i,Y}(C_{i,Z} w_{\hat{t}i,Y} + 2 w_{\hat{t}i,Z} Y - w_{\hat{t}i,Y} Z))/d \end{aligned} \tag{3.5}$$

$$d = ((C_{i,X} - X)^2 + (C_{i,Y} - Y)^2 + (C_{i,Z} - Z)^2)^{3/2} \tag{3.6}$$

Our method and its parallelization are summarized in Algorithm 1. The pseudo-code refers to the version of our implementation that uses optional statistical sampling, which is discussed further in Section 3.2.1.2. In addition, this version uses the parallelization granularity of assigning a whole CUDA thread block to triangulate each point. Further details specific to the GPU implementation are given in Section 3.2.2.

3.2.1.1 Convexity Analysis

To analyze the convexity properties of this function, we apply a practical procedure. Figure 3.2(c) shows a scalar field for Eq. 3.1 after measuring a dense set of test positions near a known ground truth position in Figure 3.2(b). The scalar field shows a very smooth variation in a large vicinity surrounding this position, where the cost has zero value. This is key since there is a high chance of convergence to the global optimum even from large distances. Such

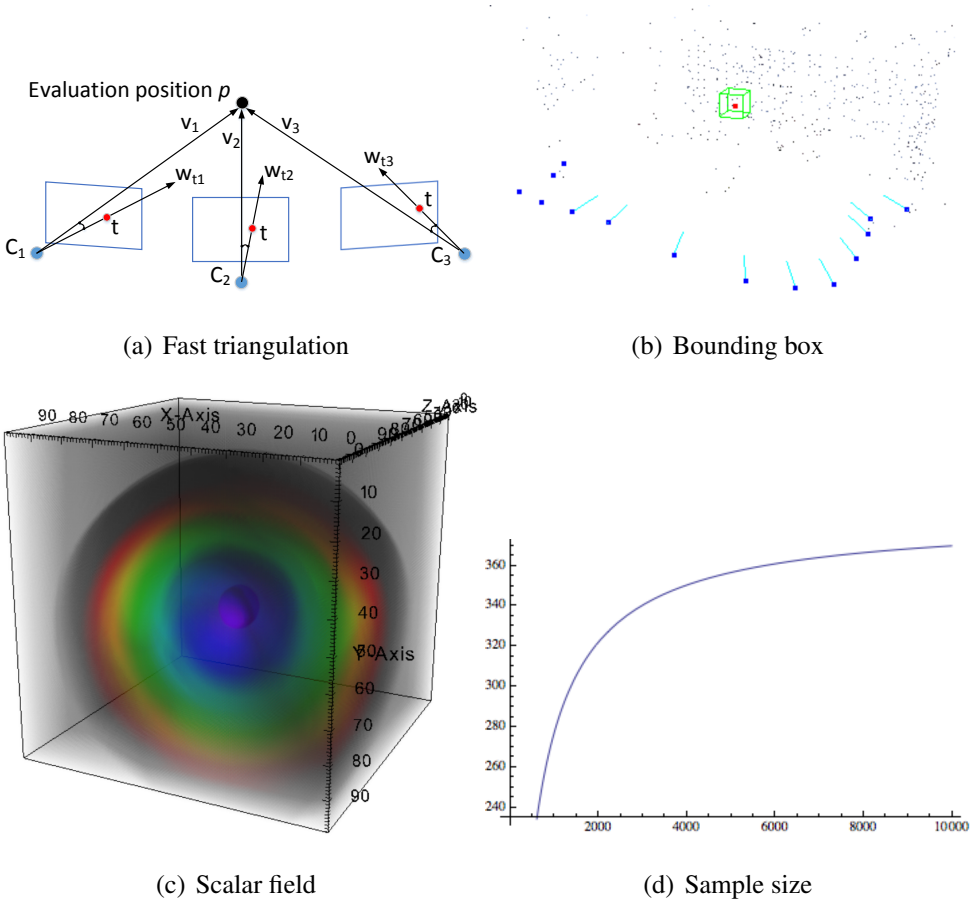
Algorithm 1 GPU Triangulation Pseudo-Code

```
1: procedure GPUTRIANGULATION(cameras, feature_tracks)
2:     for all track ∈ tracks do in parallel
3:         candidate_point ← MIDPOINTMETHOD(cameras, feature_track)        ▷ Initial guess
4:         repeat
5:             ∇cost ← COMPUTEGRADIENT()
6:             candidate_point ← candidate_point − step_size · ∇cost
7:             for all feature_i ∈ SAMPLED(feature_track) do in parallel
8:                 camera ← CAMERALOOKUP(feature_i)
9:                 v̂_i ← COMPUTERAY(camera, candidate_point)
10:                ŵ_ti ← COMPUTERAY(camera, feature_i)
11:                angular_cost_i ← 1 − v̂_i ŵ_ti
12:                angular_costs.APPEND(angular_cost_i)
13:            end for
14:            ▷ Sum all the angular costs using a parallel reduce.
15:            do in parallel
16:                cost ← REDUCE(angular_costs)
17:            end
18:        until cost < ε_threshold
19:    end for
```

scalar field renderings are not as mathematically rigorous as a direct convexity analysis, but the large basin typically seen in all of our tests indicate a strong convergence towards the global minimum. A dot product varies from $[-1, 1]$, but if we choose to deal only with points that lie in front of the cameras, the range becomes $[0, 1]$, over which the dot product is convex. A sum of convex functions is convex, as depicted in Fig. 3.2(c). Optimization is performed with adaptive gradient descent [79], starting from an initial midpoint estimate [67].

3.2.1.2 Statistical sampling

We use a statistical sampling procedure to choose a statistically meaningful sample of rays as opposed to the entire available set, N. We use Cochran's formula [16] to compute sample size n_0, as shown in Eq. 3.7. The value σ^2 is an estimate of the variance contained in the sampled data, and we used $\sigma = 0.5$ as the fixed value. Cochran's formula assumes that it is constant and known. In the general case, we do not know how far off the feature tracks are from the ground truth position, so $\sigma = 0.5$ says that these positions may vary from the ground truth with a standard deviation of ± 0.5 pixels.



(a) Fast triangulation (b) Bounding box

(c) Scalar field (d) Sample size

Figure 3.2: (a) In fast triangulation, rays are shot through a candidate point and through feature locations. (b) Multi-view reconstruction of the *castle-P19* dataset [82], with cameras in dark blue. (c) A volume view of a scalar field representing an L_1 cost function [67] evaluated at a dense grid inside a bounding box encasing a position in the reconstruction, with purple closer to zero cost. (d) Sample size (y-axis) using Cochran's formula [16] with a 95% confidence level on different population sizes (x-axis).

The value for d in Eq. 3.7 corresponds to the maximum error of estimate for a sample mean, which we fix at 5%, or 0.05. In case the obtained sample size exceeds 5% of N, Cochran's correction formula [16] should be used to calculate the final sample size, n, as shown in Eq. 3.7. In our algorithm, a 95% confidence level with a 5% margin of error is used. The variable t is the t-value, in which for a given confidence level, the corresponding percent area of a normal distribution is within t standard deviations of the mean. The t-values are typically looked up in a table, and for a 95% confidence level, $t = 1.96$. Notice in Figure 3.2(d) that the sample size stabilizes with large numbers, which is key towards our algorithm's speed.

Another option is to use RANSAC [21] to help choose a more representative sample of rays. We avoid this procedure due to iterative nature of RANSAC and its lack of an upper bound on runtime. When implementing such a method in parallel on the GPU, there could be load balancing issues because different threads could be assigned vastly different amounts of work. On the other hand, taking a single random sample of rays is fairly straightforward on the GPU, as a GPU-based random number generator can be used to select a random subset of items in an array. Furthermore, we are less concerned with outliers because our cost function is L_1 as opposed to L_2.

$$n_0 = \frac{t^2\sigma^2}{d^2} \qquad n = \frac{n_0}{1 + \frac{n_0}{N}} \tag{3.7}$$

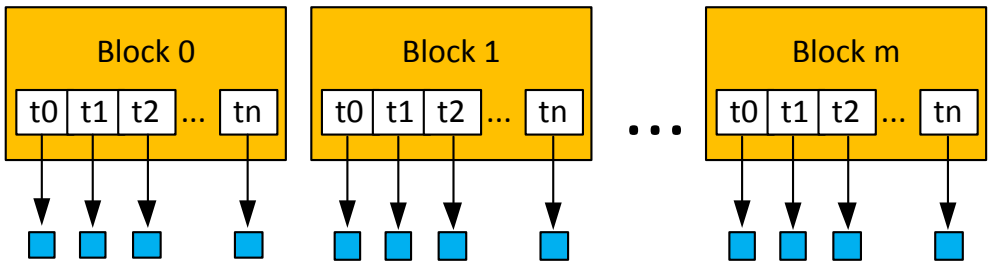
3.2.1.3 Initialization based on clustering

It is worth noting that we attempted implementing a much more robust and exhaustive initialization than that of Recker et al. [67], which is a simple midpoint start with a fixed threshold. First, the total possible number of pairs between N cameras is computed, which corresponds to $N(N-1)/2$. Next, the midpoint algorithm is used to compute a point between every possible camera pair from the sample. Then, clusering is applied on the computed midpoints. If there are no outliers, a single cluster should result. With the presence of outliers, due either to inaccurate feature tracking or a track "jumping" to a different scene point, multiple clusters could result, each of which is triangulated separately.

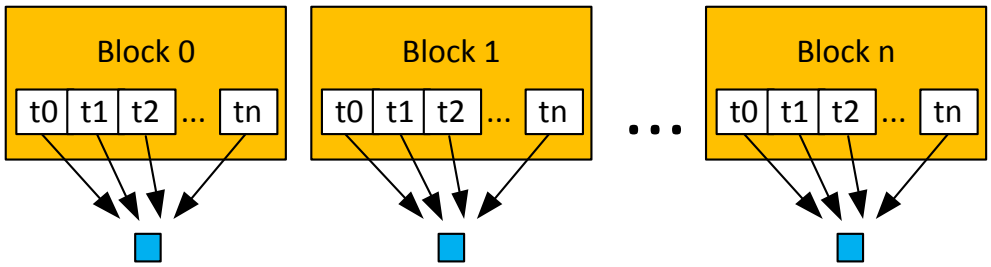
Unfortunately, this procedure leads to an order of magnitude slowdown. Also, due to the nature of the cost function and its single (global) minimum, this initialization does not lead to better accuracy. Therefore, we consider it a very important result that the original initialization method is in general better because of speed and equal accuracy than this seemingly more robust procedure.

3.2.2 GPU implementation

Our triangulation method exploits GPU properties to efficiently perform arithmetic computations derived from the L_1 cost function and its gradients. There are two main ways in which parallelization can be achieved, as discussed further in Sections 3.2.2.1 and 3.2.2.2. The simple approach is to parallelize across tracks and triangulate each track independently in a separate



(a) One thread per track



(b) One block per track

Figure 3.3: Two approaches to parallelizing our triangulator.

thread. Each thread is responsible for recomputing the gradient term for its assigned track. A second approach exploits parallelism within a track. The gradient of the cost function is computed as a sum of per-feature terms formed from the angles between rays. Instead of assigning one thread per track, an entire block of threads is assigned to each track, where individual threads compute the term for each feature in the track. The terms are then summed via a parallel reduction.

3.2.2.1 One Thread Per Track

The first implementation, shown in Figure 3.3(a), is straightforward and parallelizes across tracks, since each track can be triangulated independently of others. Each thread is responsible for recomputing the gradient for the cost function of its assigned track until convergence is reached. Considering the GPU's SIMD model, there are two drawbacks to this approach. First, some tracks may converge in fewer iterations of gradient descent than others. Second, since

different tracks can vary widely in length, as is the case in many real datasets, the gradient term may be more expensive to recompute for some tracks than for others. This creates a load-balancing issue, as threads in a warp that have finished computing its term would have to wait idly for other unfinished threads in the same warp. Some threads could perform a substantially larger amount of work than other threads.

Differing convergence rates among tracks cannot be addressed easily, as it is difficult to estimate beforehand the number of gradient steps needed for convergence. However, we can improve load balancing among threads. One way to accomplish this is already inherent in our algorithm: the use of sampling. By sampling with a 95% confidence level, an upper bound is placed on the number of features used to triangulate a track, greatly reducing track length variation since it stabilizes with large numbers. Even after sampling, however, different tracks may vary widely in length, leading to excessive thread divergence within a warp. To handle this problem, we opt to do a prior sorting of the tracks based on track length, so that threads within the same warp are likely to be assigned tracks with similar length. We can use the track lengths as integer sort keys, which allows us to use radix sort, an algorithm that maps well to the GPU [73]. We use the radix sort routine from the GPU Thrust library [7] for sorting. Sorting can reduce the divergence within warps, thereby improving performance. Figure 3.4(a) compares the performance of triangulating randomly generated, variable-length tracks with and without prior sorting. Radix sort on the GPU is fast, and we find that sorting contributes an insignificant amount of time to the overall process.

3.2.2.2 One Block Per Track

Although the GPU can support thousands of concurrent threads, individual threads have high latency. Even with sampling, a single thread that is assigned a long track could be overburdened with work. In addition, if there are few tracks, assigning one thread per track would not fully utilize the large number of available threads on the GPU. To address this, a second approach to parallelizing the triangulator assigns a block of threads to process each track. This implementation, shown in Figure 3.3(b), is more suitable for data with long feature track lengths. Each thread in a block is responsible for one feature in the gradient computation, and a parallel sum reduction produces the final gradient value for the track. Since the amount of work to compute

the gradient depends on track length, and the gradient may have to be recomputed multiple times until convergence, this approach can improve performance in long tracks. Another advantage of this approach is that it allows us to use GPU shared memory. In Kepler GPUs, each thread block has access to 48KB of shared memory. When assigning one track per thread, there is not enough shared memory to store track data for all the tracks in the thread block, even when we use sampling. Assigning an entire block to a track, combined with sampling, reduces the amount of memory needed per thread. Thus, a block's working set of track and camera data can fit in shared memory, enabling it to be used as a cache. We also perform the parallel sum reduction for the gradient in shared memory, as threads within the block must communicate to perform the reduction.

3.3 Results

The processing times and general behavior of the proposed GPU triangulator were compared against a serial CPU triangulator and a multi-core CPU triangulator on both synthetic and real data. The tested CPU was a 4-core 3.40 GHz Intel Xeon E3-1275, and the GPU was a NVIDIA Tesla K20, which features 15 SMs, for a total of 2496 cores. Tesla data center GPUs have improved performance for double-precision arithmetic, a feature we use in our triangulator. For the parallel implementation of the triangulator on the CPU, we use the OpenMP programming model to assign a group of tracks to each CPU thread. Furthermore, our CPU code uses the Eigen library for matrix and vector operations [48]. Eigen takes advantage of the SIMD units in modern CPUs (provided by SSE instructions) by using separate SIMD lanes to add or multiply more than one element in a vector or matrix for some extra parallelism. This SIMD parallelism is small, however, compared to that offered by our GPU implementation.

3.3.1 Synthetic tests

The first test on synthetic data measures the processing times as the number of tracks is increased, for the GPU implementation that assigns one thread per track vs. the multi-core CPU implementation. Track count is increased in increments of 50000, while a constant length of 100 is used for all tracks. We add image plane noise of 1% of the image diagonal dimension to the ground truth tracks, in random directions, to ensure that gradient descent requires multiple

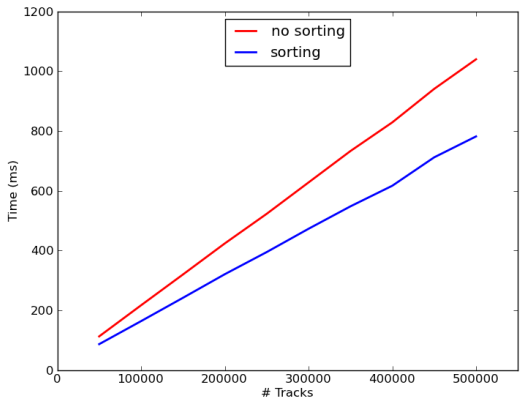
iterations to converge. Results are shown in Figure 3.4(a,b). The performance of the GPU scales better than that of the multi-core CPU as the number of tracks increases.

Next, we test GPU runtime vs track error using four types of camera configurations: *circle*, *semi-circle*, *line* and *random*. For example, in *circle*, the cameras form a circle looking at the features in the center. The *random* configuration represents a set of unstructured images such as those on the internet, where images are not acquired sequentially. Track length is fixed at 100, and the number of tracks is fixed at 10000. Figure 3.5(a) shows that runtime is hardly affected with small increases of track error.

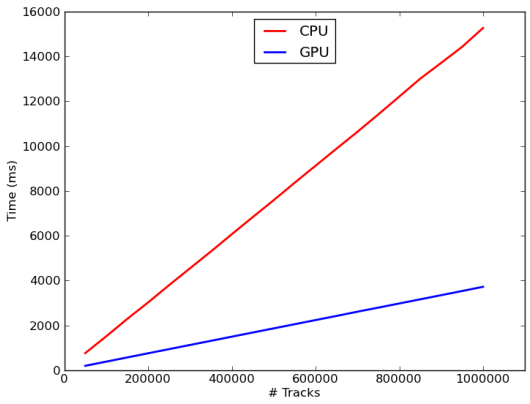
Finally, we compare the performance of the two GPU implementations. A variable track length in the range 10–100 were used, and the number of tracks tested were 20000 and 100000. In Figure 3.5(b), notice a staircase pattern for runtime in the one-block-per-thread case. This is due to the fact that a block always consists of a multiple of 32 threads (a warp). When a track length is not a multiple of 32, the extra threads are idle, so performance spikes right after multiples of 32. After 64 threads, this is no longer a problem since the fraction of idle threads in the block is small. Therefore, the performance crossover point occurs between track lengths of 32 and 64. In addition, if the number of tracks is small, the performance penalty for idling threads is small because there are fewer blocks, and therefore fewer idling threads. One-thread-per-track works better for track lengths less than 32 (a warp) and with lots of tracks. Otherwise, one-block-per-track is more scalable.

3.3.2 Evaluation on real data

For real scenes, processing time and reprojection error were evaluated, as displayed in Table 3.1. For the GPU implementation, we use the one-thread-per-track implementation with sorting because most of the tracks in the data did not exceed 100 cameras and usually had varying length, except for *Brown12*, where one-block-per-track was used. We found that the GPU implementation was at best approximately 9x times faster than multi-core and 39x times faster than serial. The general trend shows greater speedups with an increase in track count, more importantly so than the variation in track length. In the specific case of *Dinosaur* and *Stockton*, the multi-core CPU implementation is faster than the GPU, due to the fact that the track length and the number of points is too small for the GPU to make a difference. No track length is greater than 21 in

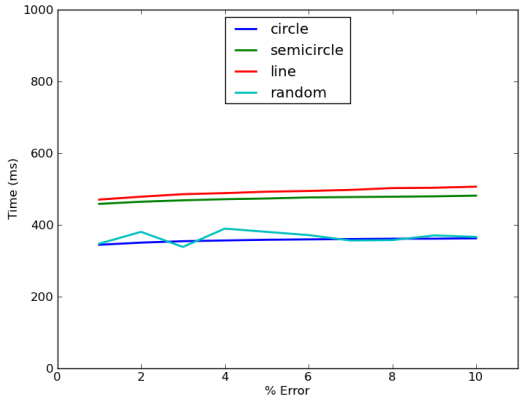


(a)

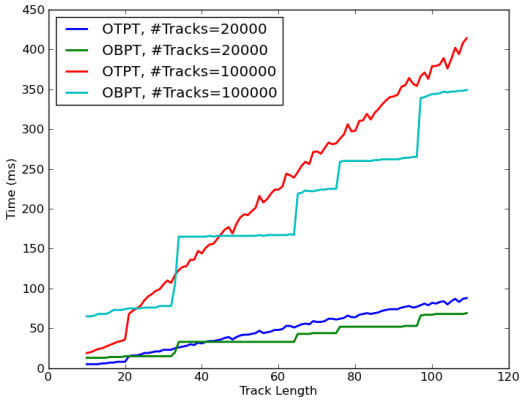


(b)

Figure 3.4: (a) GPU performance on varying track lengths (1–100) with and without sorting. (b) Performance of a multicore CPU vs. a GPU for an increasing number of tracks.



(a)



(b)

Figure 3.5: (a) GPU performance with increasing track error for different camera configurations. (b) GPU performance on varying track lengths (1–100) with and without sorting.

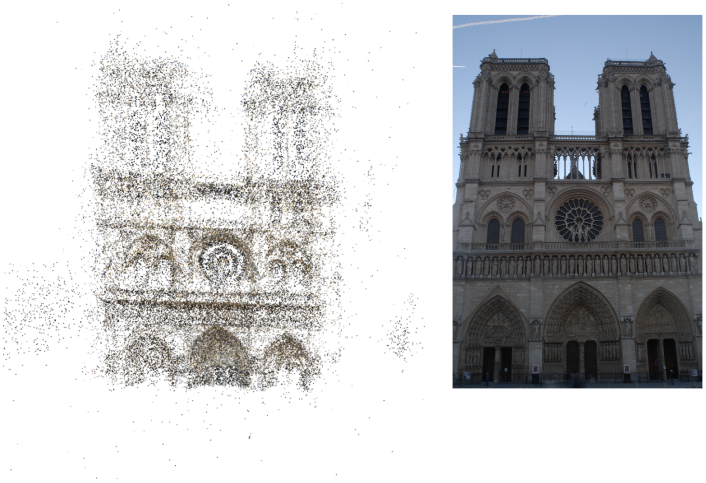
either. In contrast, in *Canyon Dense*, all tracks are of length two, but there are hundreds of thousands of them, leading to a great speedup. When comparing to other triangulators, Stewénius et al. [81] took 20 hours on the *Dinosaur* dataset [64] and Byröd et al. took 2.5 minutes, but ours takes less than 4 ms. Finally, obtained reconstructions are shown in Figure 3.6. For the Notre Dame dataset and ET dataset, shown in Figure 3.6(a) and Figure 3.6(c) respectively, the runtime results are left out of Table 3.1 due to the times being too small to be meaningful.

Data set	N	C	ϵ	t_{serial}	t_{mc}	t_{gpu}	S_{mc}	S_{gpu}
Brown12 [68]	4429	337	1.541	51	15	22	3.4x	2.3x
Dinosaur [64]	4983	36	0.467	9	2	4	4.5x	2.3x
Canyon [67]	103153	90	0.226	288	86	17	3.3x	17x
Canyon Dense [67]	997115	2	1.838	1440	342	37	4.2x	39x
Palmdale [67]	13840	66	1.159	24	8	2	3.0x	12x
Stockton [67]	16179	10	2.214	9	2	4	4.5x	2.3x

Table 3.1: Times t_{serial}, t_{mc}, and t_{gpu} and average reprojection error ϵ (pixels) with number of tracks N and number of cameras C, where S_{mc} and S_{gpu} show the speedups of a multicore CPU implementation and a GPU implementation compared to a serial implementation.

3.4 Conclusion and Future Work

This work presents a framework for GPU-accelerated N-view triangulation in multi-view reconstruction that improves processing time and final reprojection error with respect to methods in the literature. The framework uses an algorithm based on optimizing an angular error-based L_1 cost function and it is shown how adaptive gradient descent can be applied for convergence. The triangulation algorithm is mapped onto the GPU and two approaches for parallelization are compared: one thread per track and one thread block per track. The better performing approach depends on the number of tracks and the lengths of the tracks in the dataset. Furthermore, the algorithm uses statistical sampling based on confidence levels to successfully reduce the quantity of feature track positions needed to triangulate an entire track. Sampling aids in load balancing for the GPU's SIMD architecture and for exploiting the GPU's memory hierarchy. When compared to a serial implementation, a typical performance increase of 3–4x can be achieved on a 4-core CPU. On a GPU, large track numbers are favorable and an increase of up to 39x can be achieved. Results on real and synthetic data prove that reprojection errors are similar to the best performing current triangulation methods but costing only a fraction of the computation time, allowing for efficient and accurate triangulation of large scenes. Our triangulator is designed for large-scale reconstruction with ever-increasing image sizes and quantities, and opens the door for very accurate and efficient performance. Future work would pursue further performance enhancement by carefully employing both shared memory and registers to hold intermediate data and reduce global memory traffic. Additionally, recent advances in the



(a) *Notre Dame* [78] (290 views)

(b) *Brown12* [68] (337 views)

(c) *ET* [78] (7 views)

Figure 3.6: Results of our GPU triangulator alongside an image from each dataset.

CUDA programming model allow us to explore other parallelization granularities, such as using multiple thread blocks and distributed shared memory to triangulate a single point with a long feature track. This can help with load balancing, when statistical sampling is not desirable or when tracks have large variations in length. Finally, all of these potential performance enhancements would enable us to study the use of our triangulation method for real-time applications. In the next chapter, we introduce another method for triangulation that uses the path of a moving camera as a constraint and is parallelizable on the GPU.

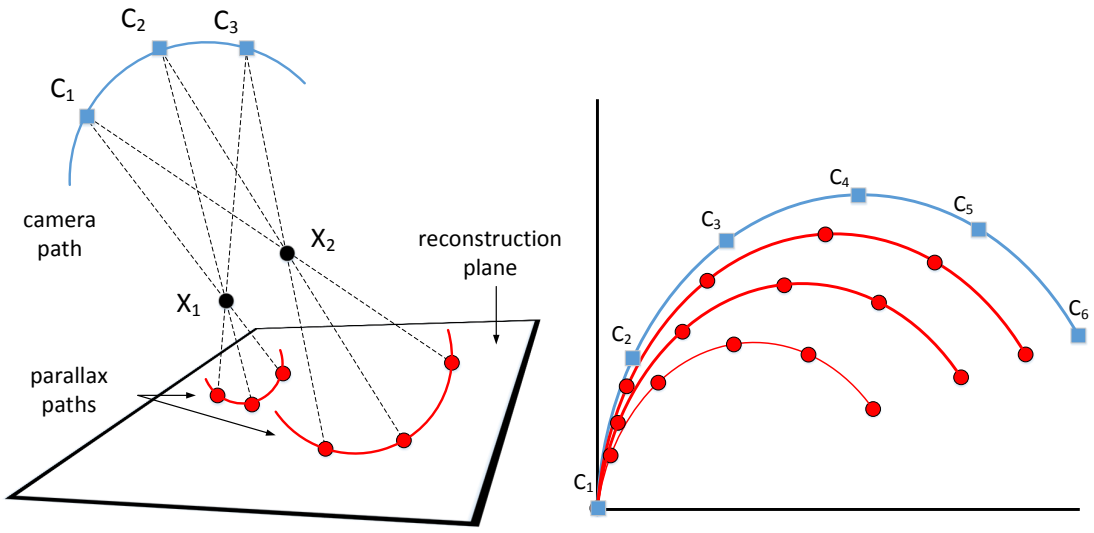
Chapter 4

Parallax Paths on the GPU

In this chapter, we map parallax paths to the GPU and test its performance and accuracy as a triangulation method for the first time. In Section 4.0, we provide an overview of the parallax paths method. Then, in Section 4.1, we briefly discuss some related work, including degeneracies in the state-of-the-art angular triangulation method. Afterwards, in Section 4.2, we describe our algorithm, go over some new insights in the parallax paths method, and present our GPU implementation. In Section 4.3, we compare our method with angular triangulation and show the results of our experiments. Finally, we end with our conclusions in Section 4.4.

4.0 Parallax Paths Definition

Recently, there is been great interest in reconstruction from aerial video. Accurate models derived from aerial video can form a base for large-scale multi-sensor networks that support activities in detection, surveillance, tracking, registration, terrain modelling and ultimately semantic scene analysis. Time-effective, accurate and in some cases dense scene models are needed for such purposes. In addition, unmanned aerial vehicles may become common tools for government and commercial use in the future, and allowing them to detect the underlying environment will enable increased autonomy and the ability to perform the type of useful analysis mentioned previously. One advantage of aerial reconstruction is the possible inclusion of sensors in addition to cameras on the reconstruction apparatus. These can include GPS and IMU, which allows for the relative camera poses to be known at the time of image capture. The additional data can act as useful constraints to improve the efficiency and accuracy of reconstruction.



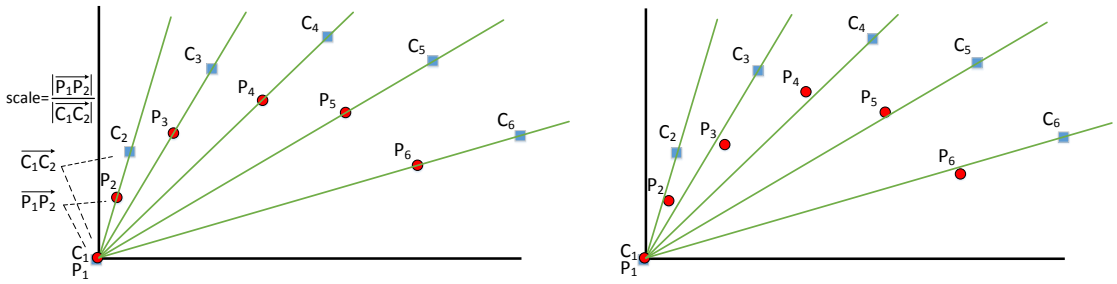
(a) The parallax paths method

(b) 2D position-invariant reference

Figure 4.1: (a) Rays from cameras $C_i \ldots C_n$ through a scene point X_i intersect a plane, creating a parallax path, which is a scaled version of the camera path. Points closer to the cameras create bigger paths. (b) The camera path and parallax paths are translated to a position-invariant reference, with a track's path origin coinciding with the anchor camera for the track.

To this end, the parallax paths method is a promising framework developed by Hess-Flores et al. [33] for aerial and turntable reconstruction. It uses the path of a moving camera as a strong constraint that can be applied to various stages in reconstruction including camera calibration, feature track correction, and final scene reconstruction. For each feature track of the reconstruction, a scale value is computed within the framework, which is a direct function of perceived parallax for the corresponding scene point. The method, however, requires that the camera path used in the reconstruction to be piecewise planar, and that it does not intersect the set of viewed scene points.

First, the method for reconstructing a single point will be summarized. It is assumed that a set of coplanar cameras and a set of feature tracks beginning at the first camera are the input. For a given feature track, a ray is shot from each camera center position through the point's pixel feature location in that camera's image plane. The intersection of this ray with a pre-selected *reconstruction plane* "beneath" the scene, which is parallel to the camera plane, yields a parallax path position. The set of all ray-plane intersections for a given feature track results in its *parallax path*. There are two insights to this method: first, if a feature track is accurate, all



(a) Perfect tracks (b) General case

Figure 4.2: (a) With perfectly correct tracks, a locus line passes through every projected feature (a path point). (b) In general, features might not lie exactly on a locus line.

rays should intersect at a common scene point; and second, the ray-plane intersections should be an exact yet scaled projection of the camera's path projected onto the plane, as shown in Fig. 4.1(a). The concept of scale is easily visualized when translating all parallax paths to a 2D position-invariant reference, as shown in Fig. 4.1(b).

Once the camera path and parallax paths are translated to this position-invariant reference, with all paths beginning at a common origin as shown in Fig. 4.1(b), *locus lines* (shown in light green) can be traced from the origin through all the parallax path points. In this case, it is assumed that the first camera is the *anchor camera* and is used as reference to provide this origin. However, any camera can be chosen as the anchor. For perfect feature tracks, a locus line should perfectly intersect every path point corresponding to a feature seen by that camera, as shown in Fig. 4.2(a). In this perfect setting, the *scale* of a parallax path is defined as the intersection between a locus line and the parallax path. Notice that scale values grow when moving from the reconstruction plane towards the camera plane.

4.1 Related Work

Like the previous chapter, the main subject of this chapter is triangulation. Related works on triangulation are summarized in the previous chapter in 3.1. The method in this chapter is within the class of algorithms relating to sequential reconstruction, where the images used for reconstruction are sequentially taken from a single, moving camera. Within this field, Pollefeys et al. [65] provides a method for reconstruction from hand-held cameras, Nistér [61] deals with

reconstruction from trifocal tensor hierarchies, and Fitzgibbon et al. [22] provides an approach for turntable sequences.

4.1.1 Degeneracies in Angular Triangulation

In the previous chapter, we looked at Recker et al.'s angular triangulation method, which we refer to from now on as *fast triangulation* [67]. There are specific degeneracies that can affect this method. The first is an initial midpoint estimate which is very inaccurate. Despite the sink behavior of the cost function, a very inaccurate starting position can lead adaptive gradient descent in the wrong direction. Though Recker et al. [67] proved that this seldom occurs, very erroneous feature tracks may need to be evaluated via RANSAC [21] or other robust methods before triangulating. The second degeneracy occurs with small baselines. For near-parallel cameras and/or small baselines, the obtained midpoint estimate can also be very inaccurate, and similar convergence issues can result. Generally, triangulating with very short baselines should be avoided, and algorithms such as frame decimation [62] can be used for this purpose.

4.2 Methodology

The parallax paths method has not been previously studied as a method for triangulation nor has it been implemented with performance in mind. In this section, we aim to explore these topics. The following are our contributions:

- We evaluate and compare the performance of triangulation based on the parallax paths framework with another algorithm used for reconstruction, Recker's angular error-based triangulation algorithm [67]. Recker's triangulation method (fast triangulation) is both accurate and one of the fastest known in the literature, and has been successfully parallellized on the GPU. This is the first comparison analysis between these two promising tools for solving the structure-from-motion problem.

- To perform the study on the most state-of-the-art high-performance hardware, we develop the first GPU implementation of the parallax paths method to compare it with the GPU implementation of the fast triangulation method.

- We further define and evaluate the effect of different path scales with respect to the original method.

- Although parallax paths requires sequential and piecewise-planar camera positions, in such scenarios, we can achieve a speedup of up to 14x over fast triangulation, while maintaining comparable accuracy.

One thing to note about parallax paths is that the method assumes the input feature tracks can be inaccurate and works to correct them. In a sense, parallax paths is not strictly a triangulation method because it alters the feature tracks themselves. However, the method does not alter the cameras and works to output the same desired result as triangulation: an accurate set of 3D scene points. The algorithm to compute parallax paths and triangulate points is outlined in Algorithm 2. In cases where there is a parallel for-loop nested inside another parallel for-loop, the maximum parallelism is the number of iterations in the outer loop times the number of iterations in the inner loop. More details about the algorithm, including the method to determine the path scales, are discussed in the following subsections.

4.2.1 Parallax Paths—A Further Analysis

The main advantage of parallax paths is that it allows for the correction of inaccurate feature tracks given constraints arising from the path of the moving camera and the projected path of a feature track as a replica of the camera path up to a scale. However, it is not clear from the original method if there is a direct way to compute accurate scale values in general, nor what the effect of scale actually is on the final computed 3D position. Also, the performance of triangulation based on the corrected tracks is not directly analyzed, and no attempts at parallelization are made. Given the way feature tracks are corrected in this method, it provides the advantages that the final triangulation can be performed efficiently, but this was not exploited by Hess-Flores et al. [33].

$$scale = \frac{|\overrightarrow{P_1P_2}|}{|\overrightarrow{C_1C_2}|} = \frac{|\overrightarrow{P_1P_3}|}{|\overrightarrow{C_1C_3}|} = \frac{|\overrightarrow{P_1P_4}|}{|\overrightarrow{C_1C_4}|} = \frac{|\overrightarrow{P_1P_5}|}{|\overrightarrow{C_1C_5}|} = \cdots = \frac{|\overrightarrow{P_1P_N}|}{|\overrightarrow{C_1C_N}|} \quad . \tag{4.1}$$

The original work by Hess-Flores et al. [33] did not mathematically define a direct way to obtain the scale of a parallax path. Here, we provide an efficient way to compute its value. For

Algorithm 2 Parallax Paths Pseudo-Code

Input: cameras, feature_tracks

Output: triangulated_points

```
1:  procedure ParallaxPaths(cameras, feature_tracks)
2:      for all i ← 0 to num_tracks − 1 do in parallel
3:          track ← feature_tracks[i]
4:          for all j ← 0 to NumFeatures(track) do in parallel
5:              feature ← track.features[j]
6:              camera ← track.camera()
7:              ray_plane_intersections[i][j] ← ComputeRayPlaneIntersection(camera, feature)
8:          end for
9:      end for
10:     for all i ← 0 to num_tracks − 1 do in parallel
11:         track ← feature_tracks[i]
12:         for all j ← 0 to NumFeatures(track) do in parallel
13:             scales[i][j] ← ComputeLocusLinesAndScale(ray_plane_intersections[i][j])
14:         end for
15:     end for
16:     for all i ← 0 to num_tracks − 1 do in parallel
17:         track ← feature_tracks[i]
18:         sum_scales[i] ← 0
19:         for j ← 0 to NumFeatures(track) do
20:             sum_scales[i] += scales[j]
21:         end for
22:         average_scales[i] ← sum_scales[i] ÷ NumFeatures(track)
23:     end for
24:     for all i ← 0 to num_tracks − 1 do in parallel
25:         track ← feature_tracks[i]
26:         correct_paths ← CorrectPaths(average_scales[i], cameras, track)
27:     end for
28:     for all i ← 0 to num_tracks − 1 do in parallel
29:         track ← feature_tracks[i]
30:         triangulated_points[i] ← Triangulate(cameras, correct_paths)
31:     end for
```

the first locus line in Figure 4.2(a), the scale of the parallax path is the ratio of the lengths of two line segments: the segment $\overrightarrow{P_1P_2}$ from the parallax path origin point P_1 and a second path point P_2; and the segment $\overrightarrow{C_1C_2}$ between the anchor camera C_1 and the next camera C_2 corresponding to the second path point. This is applicable to all the locus lines, as shown in Eq. 4.1.

The value of the ratio equals the scale of the path and is consistent for all locus line and parallax path intersections. Note that the camera path does not need to be circular or any determinable shape for this to be true, as long as all the cameras can be fitted by a common plane (are coplanar) by segments. For long camera trajectories that are non-planar, parallax paths must be computed and concatenated across segments to obtain the final reconstruction.

4.2.2 Obtaining the Correct Scale

The scale value is significant because it tells us how each feature track—and therefore each point in the reconstructed scene—relates to the camera path. In practice, there are errors in the feature tracks, and so the projected camera path or ray-plane intersections are incorrect, as shown in Figure 4.2(b). If the correct scale value for a parallax path is known, this fixes the locations of its parallax path positions along respective locus lines. For example, if we know a parallax path has a 0.5 scale of the camera path, each parallax path position on the position-invariant plot should lie halfway along the locus line segment traced between the origin and the respective camera projection. Once the correct scale value and position along locus lines have been determined, we can easily triangulate the correct 3D scene point as follows. First, the parallax path is translated back to its original position on the reconstruction plane. We then pick any two points on the path, shoot a ray from each point back to its associated camera, and compute the intersection point. This intersection is guaranteed to be unique, since all point-to-camera segments must intersect at a common 3D point given correct parallax paths, as shown in Figure 4.1(a).

In practice, there is no easy way of obtaining the absolute correct scale. However, we now propose two simple methods to approximately obtain the correct scale. The first involves averaging all the scales derived from a potentially incorrect track. In this case, the ratios in Eq. 4.1 would likely not be equal across the track, but the average of all ratios approximates the scale value. We then use this consensus scale value to correct this track. If the cameras

used in the reconstruction are too numerous, this approach could hinder performance, but we can employ statistical sampling the way fast triangulation does and use only a random subset of the features in each track to obtain an average scale. Note that there are robust methods such as RANSAC [21] that can be applied to detect highly inaccurate feature tracks. However, this adds undesired overhead to the method, and our main focus is on runtime performance.

In the second method, rather than averaging the scales derived from all cameras, or a randomly sampled subset, we only average scales for the first M cameras of the track. The reasoning behind this approach is that long feature tracks are known to sometimes experience degradation [33]. Therefore, if we assume that the first M feature track positions are more likely to be accurate, using a sequence of early cameras would yield a more accurate scale. In addition, parallax paths is less likely to suffer from degeneracy problems when the baseline between cameras is small because an intersection is enforced given the constraints no matter what the camera baseline is. However, small baselines can still introduce numerical instability for computations involving the two adjacent cameras. Therefore, care should be taken when using the first M cameras. One possible approach is to skip cameras, such as selecting cameras 1, 4, 7...etc. As long as only a subset of the cameras in the track are used, one can still achieve a fast runtime. Similar to angular triangulation, another possible approach is to use frame decimation to remove cameras from the feature track, prior to selecting a subset, so that consecutive frames have a wide enough baseline between them [62]. For our experiments on real datasets, we test the approach of using only the first 2 cameras of the track to recover the scale. Our real datasets do not have small baselines, and this test allows us to demonstrate the fastest achievable runtime.

The parallax paths method is very powerful because it provides additional constraints to yield an accurate reconstruction, which are not present in bundle adjustment [50] or traditional multi-view reconstruction. However, its application space is more limited than that of fast triangulation, since parallax paths is constrained to certain types of scenes. First, the cameras used in the reconstruction must all lie on a common plane, a case that can often be found in aerial image and turntable datasets, but that is only a subset of all possible reconstruction scenarios. Second, a proper reconstruction plane parallel to the camera trajectory must be chosen, and the

scene cannot intersect either plane. The method also needs accurate camera calibration, both extrinsics and intrinsics, since the method relies exclusively on camera information to create parallax paths and correct them. It is potentially sensitive to very inaccurate feature tracks as well. However, state-of-the-art packages such as *VisualSfM* [15] and *Bundler* [78] can provide accurate feature tracking and camera projection matrices, so this has become less of a concern. Also, accurate camera positions can be obtained from external tools like GPS and for aircraft, IMU.

As a triangulator, parallax paths can work alongside bundle adjustment to produce more accurate reconstructions. One use is to provide a good initial guess for bundle adjustment, which can increase the likelihood of converging to an accurate, optimal solution. Software such as *VisualSfM* [15] and *Bundler*, for example, use existing triangulation methods to initialize a starting point for bundle adjustment. In incremental SfM pipelines, bundle adjustment can be done multiple times throughout the reconstruction process. Like traditional triangulation methods, parallax paths can also be interleaved with periodic bundle adjustment to possibly provide better accuracy. The speed and simplicity of the method enables it to run repeatedly without a significant increase in runtime. In addition to triangulation, parallax paths can potentially be used for other purposes such as pose estimation and compression of scene information, but these are outside the scope of this work.

4.2.3 Methods on the GPU

In this section, we discuss an existing GPU fast triangulation implementation, followed by the introduction of a novel GPU implementation of the parallax paths framework, where we discuss high-level implementation details. We use the CUDA programming model to implement code and analyze performance on the GPU.

4.2.3.1 Fast Triangulation GPU Implementation

In the previous chapter, we provide a GPU implementation of Recker's fast triangulation algorithm using two different approaches. The first approach, *one-thread-per-track*, parallelizes across tracks and assigns one thread to each track to perform gradient descent for that track. This approach can potentially lead to high thread divergence. In one scenario, different tracks can vary widely in length, so the gradient term may be more expensive to recompute for some

tracks than for others. We propose that this problem can be mitigated to an extent by a prior sorting of the tracks, which increases the likelihood that threads within the same warp will be assigned tracks of similar length. Another case is when some tracks converge in fewer iterations of gradient descent than others. Both of these load-balancing problems cause threads in a warp that have finished processing their work to have to wait for other unfinished threads in the same warp. The authors also propose another approach to parallelizing the triangulator: *one-block-per-track*. This approach assigns a block of threads to process each track, which makes it more appropriate for datasets with long feature tracks. Each thread in a block computes one per-feature term in the gradient computation, and a parallel reduction sums these terms to GPU shared memory to obtain the final gradient value. In terms of the amount of parallelism during execution, this approach is an improvement over the previous.

Although the fast triangulation method obtains large speedups when run on the GPU, it still has issues fully utilizing the highly-parallel GPU programming model. The method relies on gradient descent, an iterative algorithm, making it hard to predict the amount of work needed per feature track until convergence. The step size for gradient descent must also be carefully considered due to its impact on the convergence rate and the stability of the algorithm. Furthermore, the one-block-per-track implementation can leave threads idling uselessly in a block if the track lengths are not long enough to fill a block, which must be a size that is a multiple of the warp size (32).

4.2.3.2 Parallax Paths GPU Implementation

The parallax paths method is a highly parallelizable method because the bulk of the computation involves two main stages: (1) computing ray-plane intersections for determining an initial set of parallax paths; and (2) computing all the scale values to be used in the per-track average scale. If N is the number of tracks and C is the number of cameras, there would be max $N \times C$ ray-plane intersections and $N \times C$ scale values. For computing ray-plane intersections and individual scale values, we can compute each work-item completely independently and have a maximum $N \times C$-way parallelism running on a highly-parallel GPU. In the third stage, to compute the average scales, we need to sum all the scales within each track. Although it is possible to parallelize a sum reduction, we opt to have each thread compute the sum in serial, since we only need to

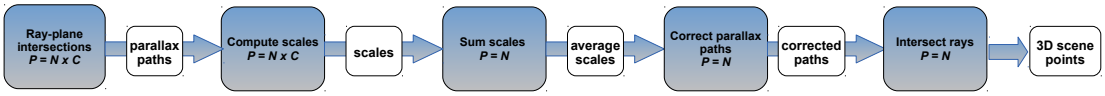
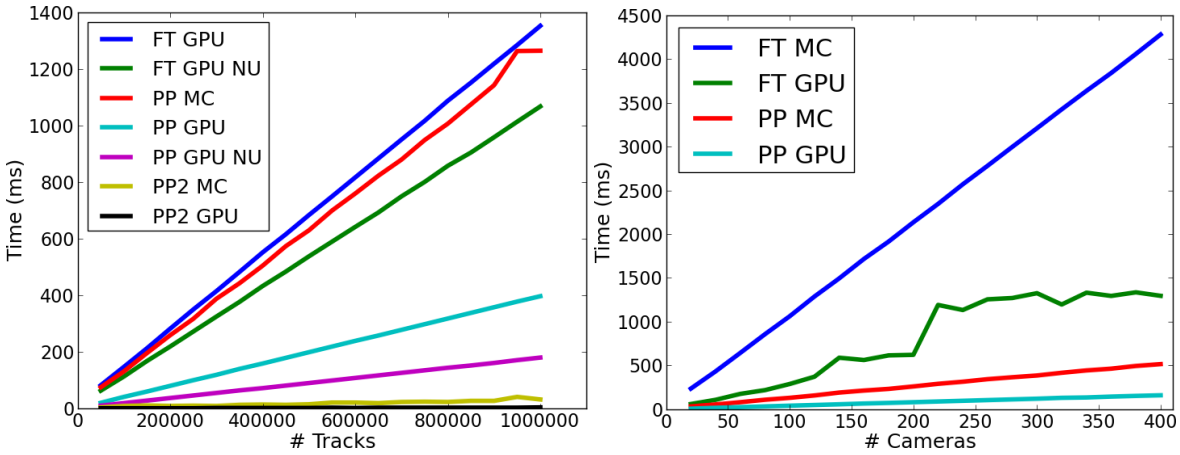


Figure 4.3: Parallax paths stages on the GPU, including parallelism P per stage.

perform the reduction once, and it is an insignificant portion of the runtime. Next, we correct the parallax path for each track. In practice, we only need to correct two points on the path because in the next and last stage, we recover the 3D position by intersecting two corrected rays from two corrected path points. Figure 4.3 shows a high-level overview of the parallax paths streaming workflow on the GPU. Although the last three stages are shown as separate, they can be combined into one GPU kernel to preserve data locality, since they all operate per track and therefore all exhibit N-way parallelism. Unlike gradient descent in fast triangulation, parallax paths on the GPU does not require multiple iterations and multiple sum reductions, instead providing a faster, more direct solution.

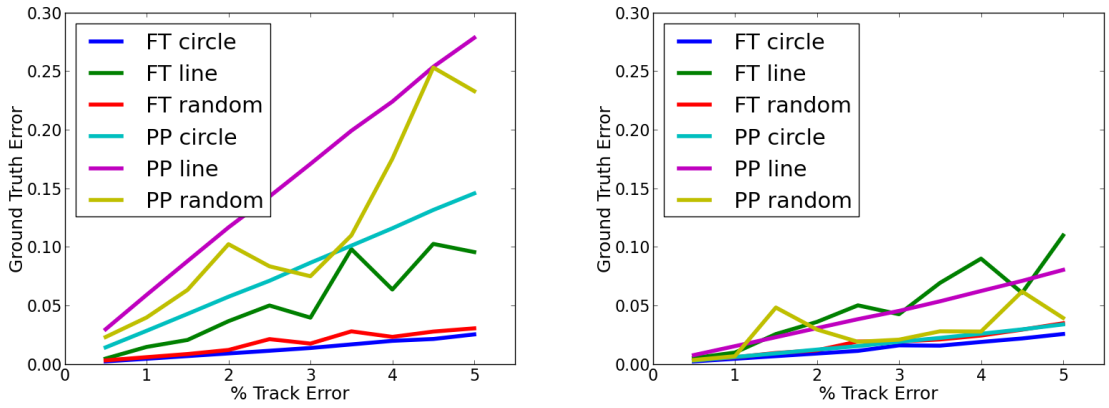
4.3 Results

We compare the processing times and general behavior of fast triangulation and parallax paths on both synthetic and real data. Our test computer has 2 Intel Xeon E5-2637 v2 CPUs, each with 4 cores clocked at 3.5 GHz, for a total of 8 cores that we use for multicore tests. Our GPU is an NVIDIA Tesla K40c, which features 15 SMs, for a total of 2880 ALUs. For running the serial tests on real data, we use a different CPU, the Intel Core i7-3630QM at 2.4GHz, since we found it had the best single-core performance. We use the OpenMP programming model to implement a multi-core parallelization of parallax paths by partitioning the set of feature tracks among the CPU threads. The following abbreviations are used throughout the section: *PP* stands for parallax paths with scale determined from an average across an entire track; *PP2* indicates parallax paths with scaling determined from only the first two features of a track; and *FT* refers to fast triangulation. *MC* denotes multi-core, while *NU* indicates that the tracks are of non-uniform length. We do not perform statistical sampling for any tests, except for some *FT* error tests, where sampling can help avoid degeneracies of close adjacent cameras.



(a) (b)

Figure 4.4: (a) Runtime performance with an increasing number of tracks. The number is increased up to 1,000,000, in increments of 50,000. Track length is fixed at 100 cameras, except for the *NU* cases, where it is varied from 2–100. (b) Runtime performance with an increasing number of cameras. Cameras are varied up to 400, in increments of 50, and track length is fixed at 100,000.



(a) (b)

Figure 4.5: Ground truth error vs. feature track error for synthetic data. (a) All features in each track subject to error. (b) No error in first feature of each track.

4.3.1 Synthetic Tests

The goal of synthetic testing is to compare fast triangulation versus parallax paths runtime performance on large-scale data and their accuracy in a ground truth sense, with ground truth not typically being available in real datasets.

Figure 4.4(a) shows runtime performance scaling with an increasing number of tracks. In this test, we use the one-thread-per-track GPU implementation of *FT*. With increasing tracks, Figure 4.4(a) shows that *PP* on the GPU scales better than its multi-core version and also scales better than *FT* on the GPU. We do not display *FT* on multi-core because its runtime is much higher than other tests. *PP2* unsurprisingly has an insignificant runtime since it only triangulates with the first 2 cameras. For the *FT NU* test, we sort the tracks to aid in load balancing. Even so, compared to *FT* on the GPU, *PP* on the GPU has a much higher improvement in runtime (a max 55% vs 22% drop) when processing non-uniform (*NU*) tracks instead of uniform tracks. The reason is that *PP* has more parallelism, with more independent work across tracks. Unlike *FT*, it does not have load-balancing issues that nullify some of the runtime reduction expected due to an overall decrease in the number of features to process.

Figure 4.4(b) shows runtime performance scaling with an increasing number of cameras. In this test, we use the one-block-per-track GPU implementation of *FT*, since it is more suitable for longer track lengths. As we discussed in Section 3.3.1, using one-block-per-track yields a staircase pattern in the runtime results, due to the warp size being 32 threads. We do not include *PP2* in these results because the method only uses 2 cameras regardless of track length. Figure 4.4(b) shows that *PP* also scales better than *FT* with increasing cameras.

In the error tests shown in Figure 4.5, we use three types of camera configurations: *circle*, where cameras were placed on a circular configuration above the scene, *line*, for a linear camera configuration, and *random*, where cameras are randomly placed in 2 dimensions above the scene while all still lying on a common flat plane. Track length is fixed at 100, and the number of tracks is fixed at 10,000.

Figure 4.5(a) shows ground truth error versus feature track error, where all features in a track are subject to error. Error is introduced to the perfect synthetic tracks by adding noise of 0.5–5% of a unit on the uncalibrated image plane diagonal in random directions. For all camera configurations, *PP* is less accurate than *FT*. Figure 4.5(b) shows the results for the same analysis, but in this case the first feature in each track (feature first seen in the anchor camera) is kept noiseless, which is a more realistic scenario. Now, we observe an improved accuracy in *PP* comparable to that of *FT*. This test demonstrates that having accurate features in anchor

frames is critical for good parallax path reconstructions.

4.3.2 Tests On Real Datasets

For real datasets, we measure performance and reprojection error, including speedup across implementations. It is important to note that the concept of reprojection error may not be applicable for parallax paths. The reason is that the camera path constraint in parallax paths enables it to be used as a means to correct feature tracks [33]. Once the scales are obtained, the tracks can be corrected and reprojected back into images, changing the features themselves and leading to a zero reprojection error. Although in the table we show reprojection error versus original feature tracks, it is not a good indicator of parallax path accuracy given that it can be forced to 0, but it's the best that can be done in the absence of ground truth information. For the tests, the real datasets were rotated to align with a vertical axis to make it easier to select a reconstruction plane for parallax paths. Table 4.1(a) displays results for fast triangulation (FT), Table 4.1(b) for parallax paths (PP), and Table 4.1(c) for PP2. For all three triangulators, larger datasets lead to larger speedups of the GPU over a serial implementation. *PP* and *PP2* are both faster than *FT*, with up to 14x and 39x speedup respectively for a meaningfully sized dataset (Canyon). However, they have higher reprojection error, though this may not be a meaningful comparison.

Finally, Figure 4.6(a)-(c) shows the reconstruction of the *Dinosaur* dataset [64] using respectively *FT*, *PP*, and *PP2* from left to right. Figure 4.6(d)-(f) displays the same but for the *Canyon* dataset [67]. For the smaller *Dinosaur* dataset, there are no obvious major differences for different methods. One limitation of parallax paths versus fast triangulation is that the scene is not allowed to intersect the plane of the cameras. To display the problems that occur, Figure 4.6(g) shows a good reconstruction obtained from *FT* for the *Horse* [58] dataset, whereas Figure 4.6(h)–(i) show the bad result obtained from parallax paths. For this scene of a horse, the camera plane intersects the top of the scene, causing some rays to be nearly parallel to the reconstruction plane, which leads to ill-conditioned problems and inaccurate reconstructed points. To obtain good parallax path reconstructions, the camera plane should be separate from the scene.

Table 4.1: Times in milliseconds for serial, multi-core (MC), and GPU with number of tracks N and total number of cameras C. Speedup is the speedup of the GPU over the serial implementation and ϵ is the average reprojection error in pixels. For the parallax path results, the speedup over fast triangulation (SU vs FT) is also shown. Some runtimes for *Horse* are left out because they were too small to measure.

(a) Fast triangulation

Data set	N	C	serial	MC	GPU	Speedup	ϵ
Dinosaur	4983	36	8	2	3	3x	0.467
Canyon	103,153	90	272	70	14	19x	0.226
Canyon Dense	997,115	2	1258	273	23	55x	1.838
Horse	9509	73	27	7	7	3.8x	0.770

(b) Parallax paths

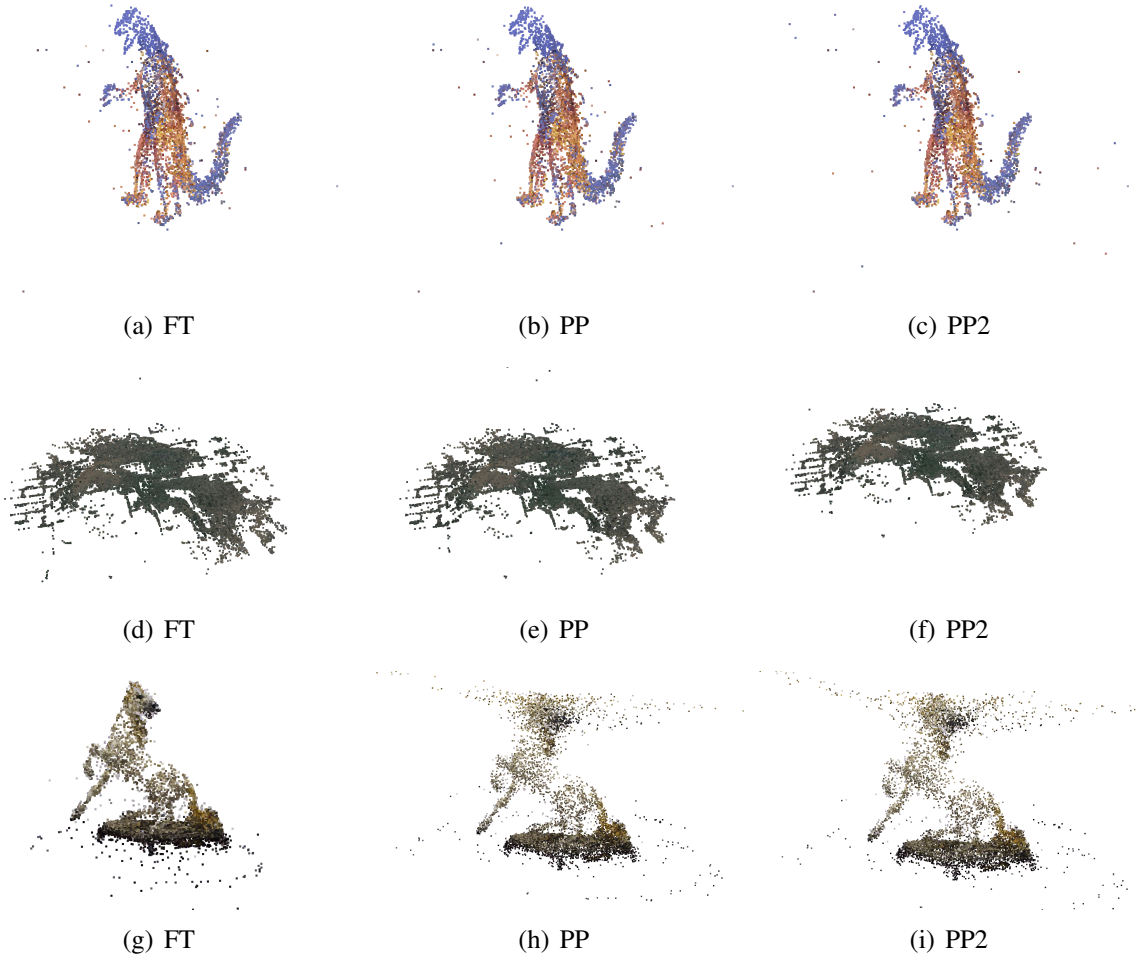
Data set	N	C	serial	MC	GPU	Speedup	ϵ	SU vs. FT
Dinosaur	4983	36	2	0.7	0.13	15x	0.668	23x
Canyon	103,153	90	75	16	1	75x	0.354	14x
Canyon Dense	997,115	2	351	64	3	117x	1.847	7x
Horse	9509	73	7	1.8	–	–	8.6	–

(c) Parallax paths first 2 cameras

Data set	N	C	serial	MC	GPU	Speedup	ϵ	SU vs. FT
Dinosaur [64]	4983	36	2	0.5	0.07	28x	1.246	42x
Canyon [67]	103,153	90	37	7	0.36	102x	0.863	39x
Canyon Dense [67]	997,115	2	351	64	3	117x	1.847	7x
Horse [58]	9509	73	3.4	0.8	–	–	1.232	–

4.4 Conclusion and Future Work

In this chapter, we present a comparison of a novel GPU implementation of a triangulator based on the parallax paths method versus the state-of-the-art multi-view triangulation method, angular error-based ("fast") triangulation. The main contributions of the work are the following. We map the parallax paths method to the GPU and analyze its performance as an efficient triangulation method for the first time. To this end, we compare it with the existing fast triangulation GPU implementation for both performance and accuracy. We make further developments to parallax paths from the original method, with more analysis on the effect of scaling. We also demonstrate the importance of having an accurate first feature in a feature track to yield an ac-



(a) FT (b) PP (c) PP2

(d) FT (e) PP (f) PP2

(g) FT (h) PP (i) PP2

Figure 4.6: Reconstructions of three scenes: (a)-(c) *Dinosaur* [64]. (d)-(f) *Canyon* [67]. (g)-(i) *Horse* [58]. Parallax paths performs poorly on *Horse* due to the camera plane intersecting part of the scene. To obtain good parallax path reconstructions, the camera plane should be separate from the scene, as is the case in *Dinosaur* and *Canyon*.

curate parallax path reconstruction. Overall, the parallax paths method is highly parallelizable and efficient, but requires that the cameras used in reconstruction be piecewise-planar and not intersect the scene itself. Though limited to applications with sequential camera motion, such as aerial video or turntable sequences, it yields a substantial speedup over fast triangulation, as demonstrated on real and synthetic testing, while maintaining comparable accuracy. If accuracy is absolutely critical, fast triangulation may still be a more preferable method. Future work involves mainly attempting to obtain more accurate scales in parallax paths, by taking into account further constraints such as intensity consensus at candidate scales. In the next chapter,

we look at a different stage of reconstruction, dense stereo, and propose a multi-GPU method that produces dense point clouds without incurring significant runtime.

Chapter 5

Efficient Dense Reconstruction on the GPU via Progressive Image Consistency Constraints

In this chapter, we introduce a multi-GPU method for creating very dense reconstructions of datasets. An overview of the dense multi-view stereo problem and existing algorithms are provided in Section 5.0 and Section 5.1. The proposed algorithm is detailed in Section 5.2, followed by results in Section 5.3 and conclusions in Section 5.4.

5.0 Dense Reconstruction Problem Definition

In this chapter, we discuss the problem of creating a dense 3D reconstruction from multiple views. Previous chapters have discussed methods that work towards obtaining a sparse reconstruction, namely feature tracking and triangulation. Figure 5.1 shows the difference between a sparse and a dense reconstruction. The sparse reconstruction, however accurate, is not complete and contains "holes", which may be beneficial to fill in. The goal is to obtain a dense structure that is both complete and accurate.

In the past several years the number of applications that benefit from dense and efficient multi-view reconstruction has surged. Robotics, for example, has experienced an increased interest in quadcopters and other personal drones due to their increased affordability. Similarly, consumer attention to 3D printing has symbiotically led to increased 3D scanning needs. Both

Figure 5.1: An initial sparse reconstruction, with good camera estimation, on the left. It can be made into a very dense version, on the right, by our method.

of these areas, and classical applications such as surveillance and terrain modeling, are made possible by efficient, accurate, and dense modeling of the scenes and objects in their view. State-of-the-art algorithms [26, 78, 90] are based mainly on sparse feature detection and matching, utilizing the Scale-Invariant Feature Transform (SIFT) algorithm [51] and other feature trackers inspired by its concept. For general scenes, these algorithms provide reasonably accurate feature tracking, camera poses, and scene structure.

However, a key observation is that most of the data for the previously mentioned applications consists of sequential images of a scene whose geometry varies gradually along a gradient, i.e., there are few sharp jumps in depth on a per pixel basis. Many prior algorithms have taken this into account by doing region-based calculations [37, 86–88, 96]. Our proposed method revisits this work by creating regions with a modern superpixel algorithm, SEEDS [84], interpolating iteratively for higher fidelity, and performing computation on multiple GPUs. Specifically, it consists of a two-phase algorithm that iteratively solves for unknown pixels by interpolating and optimizing ray-distance values using a multi-image consistency check via the Colored SIFT (CSIFT) descriptor [1]. The accuracy of the resulting reconstruction is only limited by the accuracy and density of the initial known reconstruction and the accuracy of the camera parameters. For example, the *dinosaur* dataset has very accurate camera positions and an initial reconstruction produced using SIFT features can be made very dense via our method as shown in Figure 5.1.

5.1 Related Work

There are a quite a number of dense reconstruction algorithms in the literature [14, 20, 23, 24, 27, 32, 37, 41–43, 53, 69, 85, 87, 88, 94, 96]. Perhaps the best known of these algorithms is *Patch-Based Multi-View Stereo* (PMVS) [24]. This algorithm creates quasi-dense reconstructions by enforcing photo-consistency constraints on patch matching. The upgraded *Clustering Views for Multi-view Stereo* (CMVS) version [23] provides higher efficiency by intelligently grouping sets of images, and does not have memory limitations, but still suffers from non-completeness and a lack of additional constraints. Both algorithms are part of the popular *VisualSfM* program [90] for 3D reconstruction. There is even a further improvement, *Tensor-Based Multi-view Stereo* (TMVS) [92], which suffers from the same problems.

Besides patch-based multi-view stereo, there are a number of image-based rendering methods in the literature and others that can provide a fully dense and watertight reconstruction. Examples of this, in order, are shape from silhouettes [83], voxel coloring [76], and space carving [44, 46]. In summary, shape from silhouettes [83] is a form of voxel labeling, in which the visual hull of the viewed shape is computed by intersecting the projected volumes of the object's silhouettes as they appear in each input image. Voxel coloring [76] differs in that it computes a photo-consistent 3D shape by voxel projection followed by correlation of pixel colors amongst the input images. Space carving [44, 46] uses a multi-pass sweep of a plane to eliminate voxels that violate the photo consistency constraint, as does plane sweeping [25]. The main issue with these algorithms is that they typically rely on an accurate knowledge of the viewed object's silhouette, and thus have a more restricted application space than multi-view stereo methods, which do not have this requirement. Furthermore, since plane sweeping is based on homographies, there could be "drifting" of the obtained feature tracks, and inaccuracy in the 3D points. An advantage of space carving is that it doesn't depend on texture or color, and is capable of producing a dense, water-tight reconstruction by virtue of the approach. However, because of inaccuracies in the obtained silhouettes or input camera parameters, it is seldom accurate enough to capture very fine details.

There also exist a number of volumetric methods [41, 42, 85]. Given the recent advances in convex optimization, globally optimal formulations have been proposed for the multi-view

reconstruction problem [41, 42]. However, this line of research has so far mainly focused on the optimization methods themselves. In order to obtain highly accurate reconstruction results, the data term in energy formulations is just as important. Even the best currently available approaches have major problems in low-textured image areas, leading to visible artifacts in the obtained reconstructions. Kostrikov et al. present a formulation based on an analysis of why volumetric approaches have problems in specific challenging regions [43]. They provide a probabilistically well-founded formulation for the labeling cost that is more robust to outliers and achieves improved reconstruction results. Though they obtain great results, their method uses an outlier removal step that affects completeness and is still based on the use of a cost function.

The Middlebury Multi-View Stereo Evaluation provides a benchmark for comparing dense reconstruction algorithms [77]. This evaluation is based on completeness percentage and accuracy. According to the results, which are updated live through user input, PMVS/CMVS is still the top performing method overall as far as completeness, with the method by Guillemaut and Hilton [27] also performing very well. It is hard to see a clear trend in accuracy; both those methods plus Kostrikov et al. [43] perform well in most evaluations. As far as runtimes, most methods take several hours on the tested datasets, when runtimes are normalized to a 3.0 GHz processor frequency. By far, the fastest overall are the methods by Zach [94], Merrell et al. [53] and Chang et al. [14], which take on the order of just a few seconds, but have a lower accuracy and completeness percentage overall than the top-performing methods in those categories.

Given the problems with the current literature, it is desirable to find a method that is accurate, dense, efficient, and does not require additional image information, such as silhouettes. As will be described, this can be achieved with a deceptively simple, highly parallelizable method that isn't far from a brute-force algorithm.

5.2 Methodology

We introduce a brute-force method for creating very dense reconstructions. Our densification algorithm can begin with any initial reconstruction. The key behind the algorithm is that the distance along the ray from the position of the 3D structure to the camera (ray-distance) varies

smoothly for image regions corresponding to the same object. Furthermore, these regions do not have to be computed on a per-object basis since objects can be over-segmented by a superpixel algorithm [84]. We make the following contributions:

- We introduce a novel two-pass algorithm that interpolates depth values in two-dimensional image space within a superpixel region and then optimizes the interpolated value via image consistency analysis across neighboring images in the dataset.
- Our method is modern in many ways, including its use of region segmentation via the SEEDS SuperPixel algorithm [84] and its use of an effective and fast descriptor, CSIFT [1].
- Our method parallelizes well on a GPU. For the Middlebury Temple dataset, our proposed GPU-accelerated method outperforms PMVS/CMVS in runtime (1.5 minutes vs. 4.5 hours).

The details of the algorithm are presented in Section 5.2.1, along with GPU implementation details in Section 5.2.2.

5.2.1 Densification Algorithm

The goal of our densification procedure is to find a ray-distance value for each and every pixel in the image sequence. This value, along with the pixel coordinate, uniquely defines the 3D location in the scene for that pixel. Our algorithm depends on an initial reconstruction (input images, camera projection matrices, and initial 3D structure), for which these distance values have been computed. We compute a SuperPixel segmentation [84] of the input images to roughly segment each image into regions corresponding to the same objects. Our algorithm then utilizes two major procedures, (*interpolate* followed by *optimize*).

During interpolation, we use known ray-distances from the initial reconstruction to interpolate a distance for the given pixel being solved. This distance is used as a starting point for optimization, where multiple distances are tested for the best image consistency after reprojecting back into the images. After the distance is optimized, we triangulate the point and add the computed distance to the set of currently known distances. In addition, we add the triangulated 3D point to the reconstruction. This process can take one or more iterations depending on the

density of the initial reconstruction. Pseudo-code for the densification algorithm is shown in Algorithm 3.

Algorithm 3 Densification Algorithm

```
1:  procedure DENSIFY(images, init_recon)
2:      for each image ∈ images do
3:          known_dists ← COMPUTEKNOWNDISTS(init_recon)
4:          for each superpixel ∈ non_empty_superpixels do
5:              for each pixel ∈ superpixel do
6:                  guess_dists ← INTERPOLATE(known_dists, pixel)
7:                  dist ← OPTIMIZE(guess_dists, images)
8:                  current_dists. APPEND(dist)
9:                  point ← TRIANGULATE(dist)
10:             end for
11:         end for
12:         known_dists ← UPDATEKNOWNDISTS(current_dists)
13:         while unsolved superpixels remain do
14:             for each superpixel ∈ empty_superpixels do
15:                 neighbor_super_pixels ← GETNEIGHBORSUPERPIXELS(superpixel)
16:                 for each pixel ∈ superpixel do
17:                     guess_dists ← INTERPOLATE(known_dists, neighbor_superpixels, pixel)
18:                     dist ← OPTIMIZE(guess_dists, images)
19:                     current_dists.APPEND(dist)
20:                     point ← TRIANGULATE(dist)
21:                 end for
22:             end for
23:             known_dists ← UPDATEKNOWNDISTS(current_dists)
24:     end for
```

5.2.1.1 Interpolation

The idea is that pixels within the same superpixel likely belong to the same object. Therefore, in the first iteration, we can interpolate pixel distances using pixels within the same superpixel for which the distances are known. This can help alleviate computational runtime by producing a better starting point for each pixel and allowing for a smaller search space during the next stage. For our datasets, we find that the initial sparse reconstructions are dense enough so that there are no initial empty superpixels (superpixels in which there are no pixels with solved distances). In the event that there are empty superpixels, our algorithm can enter another phase, where pixels with known distances in neighboring non-empty superpixels can be used to interpolate the dis-

tances for the unsolved pixels. This can affect accuracy and/or increase computational runtime, if the resulting interpolation is not an accurate starting point for the next stage. However, it guarantees that all superpixels will eventually be solved.

5.2.1.2 Optimization

The *optimize* function fine-tunes the ray-distance estimate from the interpolation procedure for the given iteration. This procedure defines a small ray-distance search space around the given interpolation estimate to search for the best possible value. The best distance is determined using image information from a window of neighboring images in the image sequence. The effect of window size is explored in Section 5.3.1.

Specifically, the search space is set to the initial distance estimate plus/minus a user-defined threshold. This space is quantized into n candidate distances. The density of this quantization is limited by the desired computation time. For each candidate ray-distance, the corresponding feature track is first computed for all images within the window. This is accomplished by simply traversing the ray generated from the pixel location for the candidate ray-distance. In other words, this procedure searches along epipolar lines in the image sequence.

To ensure the best possible match is obtained, the CSIFT [1] descriptor is evaluated at each feature track location for each candidate ray-distance. This evaluation is essentially checking for image consistency for a given pixel's candidate structure point. The given pixel provides a reference CSIFT descriptor, c_{ref}. The CSIFT descriptors at the feature track locations, c_i, (generated by the candidate ray-distances) should be very similar. The algorithm chooses the candidate ray-distance that minimizes the L_1 Euclidean distance between c_i and c_{ref}, for all i, across all the feature tracks generated from candidate values. Notice there could be occlusions present at the correct distance; images at which these occur increase the error value. However, the total error is not as high as in cases where the wrong distance is being evaluated since *most* images will coincide.

Naturally, this optimization scheme works well when the camera parameters are perfect or very accurate, and the desired feature is actually present on the epipolar line along which the optimizer searched. To account for datasets where the camera parameters are not completely accurate, the search can be expanded to include a variable number of lines parallel to the initial

epipolar line. The less accurate the camera parameters are, the more extra lines should be searched, but this increases the likelihood of false positives. A good number of extra lines to search could be congruent to the reprojection error of the initial reconstruction (i.e., 2 pixels of reprojection error should require searching 2 extra lines above and 2 extra lines below).

5.2.1.3 Cleaning

As one might imagine, there may be a considerable amount of noise produced in failure cases and also a large amount of redundancy. Luckily, since the algorithm has redundantly added the "same feature" to the image multiple times (as many times as it appears in unique images), noise can easily be removed by performing a statistical outlier removal and then merging close points to minimize the size of the reconstruction. This optional functionality was implemented using the Point Cloud Library [70]. Additionally, it might be necessary to perform background segmentation on images from a turntable dataset to avoid reconstructing background pixels.

5.2.2 GPU Implementation

The method solves for the depth of every pixel in every image, a computation workload that can be performed in parallel. Our implementation works on multiple GPUs, as the pixel-distance pairs to solve can be split independently among multiple GPUs. OpenMP is used to manage the separate GPUs.

Between the two major stages of our method, *interpolate* and *optimize*, the more expensive is *optimize*. During this stage, hundreds of candidate ray-distances for each pixel are tested in the search for the most optimal one. For each candidate ray-distance, a descriptor is computed on the neighborhood patch surrounding the coordinates where the candidate ray-distance is projected onto an image. The absolute difference (L_1 Euclidean distance) between this descriptor and the reference descriptor is computed, and the smallest average of differences across all images in the image window determines the final depth that is most optimal. For simplicity, the optimization of each pixel-distance pair is assigned to one CUDA block. Each thread computes multiple descriptors, one for each image in the image window, and saves the average difference of the descriptors with respect to the reference descriptor. Next, finding the smallest average difference across threads is simply a blockwise parallel reduction that is done in shared memory. This reduction is implemented using the NVIDIA CUB library [55]. Figure 5.2 illustrates

the work granularity of searching for the optimal depth of a single pixel.

Traversing the ray-distance for a pixel in the reference image corresponds to traversing an epipolar line in another image. This approach of searching for the best depth leads directly to descriptors being computed along diagonal epipolar lines in the images. Although image data is stored linearly in memory, this cannot be exploited due to a lack of predictable regular memory access patterns when reading along diagonal epipolar lines. In the work of Iandola et al., the authors test the performance of image convolution kernels using different GPU memories [35]. For Kepler GPUs, the authors find that loading image values from texture memory to registers and performing the convolutions in registers yield speedups of 1.9x to 8.8x when compared to a naive approach that only uses global memory. They also test a shared memory implementaton, which does not achieve good performance. Although our problem is not exactly the same, we find their results meaningful because we also process regions of an image in parallel. With this in mind, we choose to store our images in texture memory instead of global memory. A Kepler GPU's dedicated hardware for texture fetching and its texture cache are efficient for irregular groups of memory reads, as long as the reads have spatial locality. Adjacent GPU threads in our implementation compute descriptors along the same epipolar line, enabling the memory accesses to have 2D spatial locality. Additionally, descriptors are computed with image data at a subpixel level, and texture memory is optimized for fast subpixel interpolation.

5.3 Results

The proposed algorithm was analyzed for its general behavior and processing time on several real datasets. The effect of window size in the optimization routine was tested on datasets with ground truth information available. The algorithm was implemented in C++, parallelized with CUDA, and all results were generated on an Ubuntu 12.04 Linux machine with an Intel Xeon E5-2637 and four K40C NVIDIA GPUs.

5.3.1 Window Size Justification

In the optimization procedure, we perform a photo-consistency check across neighboring images in the sequence. Therefore, we find it necessary to experiment with the amount of neighboring images (the window size). A window size of two refers to analyzing one image before

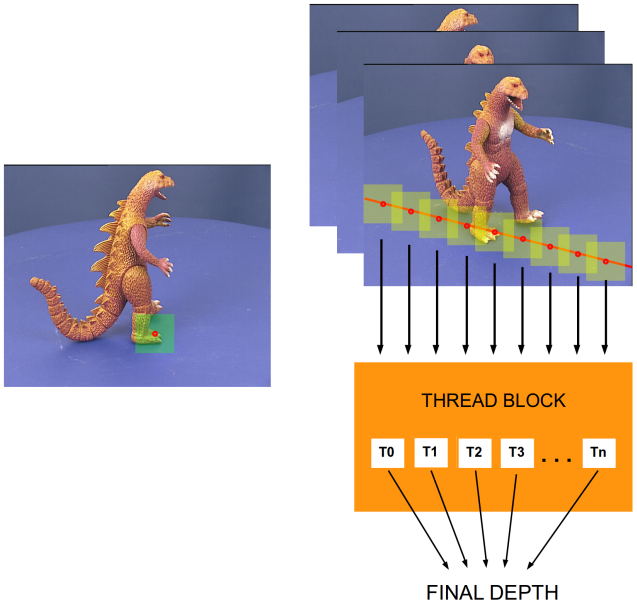


Figure 5.2: A pixel whose depth is to be optimized is highlighted on the left. A reference descriptor is computed at this pixel. Candidate depths are projected to other images in the window of images. As the depth is varied during the search, its projection to other images traverses epipolar lines. Descriptors are computed on the projected points, and the L_1 Euclidean distance between each descriptor and the reference descriptor is computed. The work to compute descriptors and distances are assigned to threads within a block. A final block-wise reduction determines the smallest average distance and the best depth.

and one after the current image, a window size of four corresponds to analyzing the two images before and two after, and so on. First, we generate ray-distances using the ground truth 3D information and camera data. Next, we compute the standard deviation, σ, of the ground truth ray-distance values, s_i. To analyze the robustness of the *optimize* procedure, we introduce noise into the ground truth distances at varying levels, $\epsilon = [0, 1, 2, 3]$, and then we execute the *optimize* procedure using the noisy estimate, s_i' and a specified window size. We compute the noisy estimate by sampling a uniform distribution in the range of $s_i' \in [s_i - \sigma \times \epsilon, s_i + \sigma \times \epsilon]$. Reported error values correspond to average distance error between computed and ground truth 3D points, for a full reconstruction. Table 5.1 shows the results of the test when run on the ground truth Oxford Dinosaur dataset [64]. For all tested window sizes, and low noise levels $\epsilon = 0$ and $\epsilon = 1$, we obtain similar error values overall. Therefore, using a window size of two is usually justified since it is less expensive to compute and provides the same results. For larger window sizes, there is a risk of running into occlusions and other wide baseline effects that might af-

fect scoring. For high noise levels, such as $\epsilon = 2$ and $\epsilon = 3$, errors were significantly higher, and relatively constant across window sizes. For our results on sequential images, we choose a window size of four to keep runtime small, while still obtaining a sufficient photo-consistency check.

Table 5.1: Average 3D positional error at varying window sizes and noise levels.

Window size	2	4	6	8	34
Score $\epsilon = 0$	0.0013	0.0014	0.0016	0.0018	0.0093
Score $\epsilon = 1$	0.0014	0.0015	0.0017	0.0019	0.0086
Score $\epsilon = 2$	0.0287	0.0236	0.0207	0.0191	0.0239
Score $\epsilon = 2$	0.0815	0.0682	0.0612	0.0584	0.0758

5.3.2 Results on Real Datasets

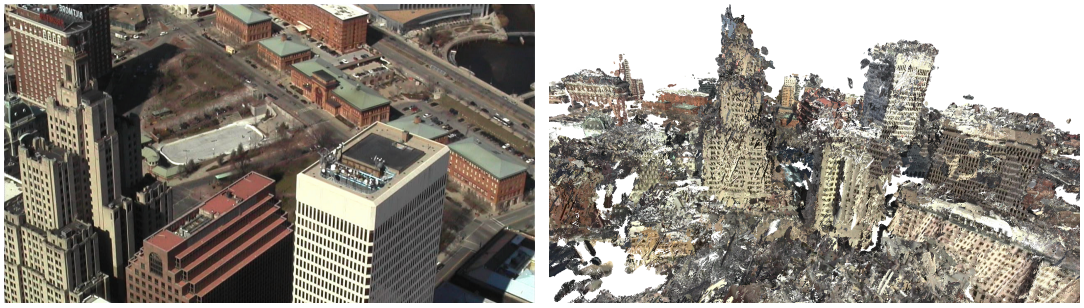
To analyze the efficacy of the algorithm, the densification procedure was executed on the Middlebury Temple [77] dataset benchmark. Results, as displayed in Figure 5.3, show that the densification procedure outperforms PMVS/CMVS in runtime (1.5 minutes vs. 4.5 hours). We also found them to be similarly complete.

In addition, we tested our method on a dataset with aerial images, since aerial scenes can often resemble turntable sequences. Figure 5.4 shows a reference image from the Brown Site 22 dataset [13], and a dense reconstruction of the scene produced by our method. As shown in the figure, our result is noisy and unable to accurately capture many details in the scene. Like many other algorithms, our method struggles with aerial scenes due to their inherent challenges, including the difficulty of estimating accurate camera parameters and the low resolution of the images with respect to the relatively large scale of the scene in real life. The latter prevents our method from performing accurate region segmentation. However, considering the size of the images in the dataset (1280x720), we achieve a reasonable runtime of 32 minutes. We believe our method, with its use of GPUs, can enable the reconstruction of very large-scale aerial scenes. In the future, we hope to explore ways to improve accuracy, particularly by incorporating other sensor data, such as GPS, in the process to recover accurate camera parameters.

As discussed earlier, one limitation to the final correction stage of the algorithm is that it can only be used for sequential image streams, unlike PMVS/CMVS which is more general.

(a) Reference Image (b) Our Result

Figure 5.3: (a) Reconstructions of the Template Dataset. Reference image of the Temple dataset. (b) Complete dense reconstruction of the Temple dataset via the proposed method. For this result, 39 images of size 640x480 were used. The result of the proposed method outperforms PMVS/CMVS in runtime (1.5 minutes vs. 4.5 hours) and is similarly complete.



(a) Reference Image (b) Our Result

Figure 5.4: The result of our method on Brown22, an aerial dataset containing 243 images of size 1280x720.

However, there are a great number of relevant problems in sequential reconstruction, spanning many important applications, where an accurate and dense reconstruction is necessary, and the proposed algorithm is capable at meeting these requirements. Finally, Figure 5.5 shows a number of dense reconstructions obtained with the proposed method.

5.4 Conclusion and Future Work

This work presented an updated, efficient take on an old deceptively simple algorithm. It was modernized in many ways, including region segmentation via the SEEDS SuperPixel algorithm [84], the use of an effective and fast descriptor, CSIFT [1], and parallelization on the GPU. Additionally, it proposes a two phase approach that allows for even denser reconstruc-

Figure 5.5: Results from the proposed method for the Dinosaur and Conch datasets. The first column shows an example input image, the second shows the initial sparse reconstruction used as input, the third shows results from the proposed method, and the last column shows the result of CMVS/PMVS as implemented by VisualSfM. For the Conch shell dataset with 216 images of size 640x480, our method takes 3 minutes. For the Dinosaur dataset with 36 images of size 720x576, our method takes 1.3 minutes.

tions when initial reconstructions are well distributed in 2D space.

Ideally, future improvements on this algorithm would lower runtime by exploring data structures to enable more efficient memory access patterns on the GPU. Reconstruction of aerial scenes can be revisited by incorporating more sensor data. Accuracy improvements can also be attempted by applying geometric constraints from known geometries (such as the rectangular nature of buildings). In the next chapter, we present a divide-and-conquer method to parallelize bundle adjustment, an expensive stage in reconstruction, on multiple GPUs.

Chapter 6

Parallel Bundle Adjustment

In this chapter, we study the implementation of bundle adjustment, the most expensive stage in 3D reconstruction in terms of performance, on multi-GPU systems. First, we give an overview of the bundle adjustment problem, including approaches on the GPU, in Section 6.0. Next, we discuss related work in Section 6.1 before giving details of our methodology in Section 6.2. We present results in Section 6.3 and end with conclusions in Section 6.4.

6.0 Problem Definition

6.0.1 Bundle Adjustment

Bundle adjustment is a key step of most SfM systems. At minimum, the algorithm is usually run as a final step in the SfM pipeline. Often, it may be done multiple times periodically in the pipeline. A bundle adjustment problem is defined by a set of cameras, 3D points, and the tracking of these points across the cameras. These form a non-linear optimization problem, in which the cameras and points are refined to minimize reprojection error. Due to the accumulation of error throughout an SfM pipeline, bundle adjustment helps keep the reconstruction accurate and consistent. The following is the mathematical formulation of the optimization problem:

$$\min_{c_j, p_i} \sum_{i=1}^{n} \sum_{j=1}^{m} v_{ij} d(y_{ij}, f(c_j, p_i))^2 \tag{6.1}$$

Here, $f(c_j, p_i)$ is the predicted projection of point i on image j, and y_{ij} is the observed projection of point i on image j. The term $d(y_{ij}, f(c_j, p_i))$ is the distance between the predicted

and observed projection, while v_{ij} denotes visibility and equals 1 if point i is visible in image j and 0 otherwise. The equation can be further simplified by first combining the parameters c and p into a single parameter vector β. Next, $d(y_{ij}, \beta)$ is expanded as the distance formula, and the squaring of each term cancels out the square root in the formula. In the new equation, each term is the square of the difference in 2D image space between the predicted projection and the observed projection in either the x or y direction. The visibility term v_{ij} is left out for simplicity.

$$\min_{\beta} \sum_{i=1}^{n} \sum_{j=1}^{m} (y_{ij1} - f(\beta)_1)^2 + (y_{ij0} - f(\beta)_0)^2 \tag{6.2}$$

With a sum of squared differences, Equation 6.2 formulates a non-linear least squares problem for model fitting. The general form of such problems can be written as:

$$\min_{\beta} \sum_{i=1}^{m} (y_i - f(\beta))^2 \tag{6.3}$$

Solving such problems requires the use of iterative approaches. For bundle adjustment, the algorithm that typically obtains the best performance is Levenberg-Marquardt [47]. This method is within the class of algorithms known as trust-region methods and is implemented in the popular open-source software, Ceres Solver [3]. The method assumes that within a certain region, a model (usually quadratic) of a function is a good approximation of the function. A candidate step is evaluated with the goal of moving closer to the function's minimum. If the step is deemed poor (taking it actually increased the function value), the step is not taken, the trust region is shrunk, and a new step is considered.

In each iteration, the parameter vector β is replaced by a new estimate $\beta + \delta$. The function $f(\beta + \delta)$ can be approximated by its linearization:

$$f(\beta + \delta) \approx f(\beta) + \boldsymbol{J}\delta \tag{6.4}$$

Here, $\boldsymbol{J}$ is the Jacobian of f. The approximation can be substituted into the objective func-

tion in the non-linear least squares problem:

$$S(\beta + \delta) = \sum_{i=1}^{m} (y_i - f(\beta) - \boldsymbol{J_i}\delta)^2 \tag{6.5}$$

The minimum of this function occurs where the first-order derivative equals zero. Taking the derivative with respect to δ and setting the result equal to zero yields the equation:

$$(\boldsymbol{J}^T\boldsymbol{J})\delta = \boldsymbol{J}^T(y - f(\beta)) \tag{6.6}$$

This new formulation gives the *normal equations*. During each iteration, the linear system is solved to obtain δ. The size of the linear system depends on the Jacobian matrix, which in turn depends on the number of cameras and points in the scene. Due to the nature of bundle adjustment, which optimizes two distinct sets of parameters (those for the cameras and points), the derived matrix of the linear system can be divided relatively easily into four sub-matrices. This in turn allows the use of the *Schur Complement* trick to solve the linear system, which produces a smaller system of equations. Once the reduced system is solved, the rest of the system can be solved using a trivial back-substitution.

6.0.1.1 Schur Complement

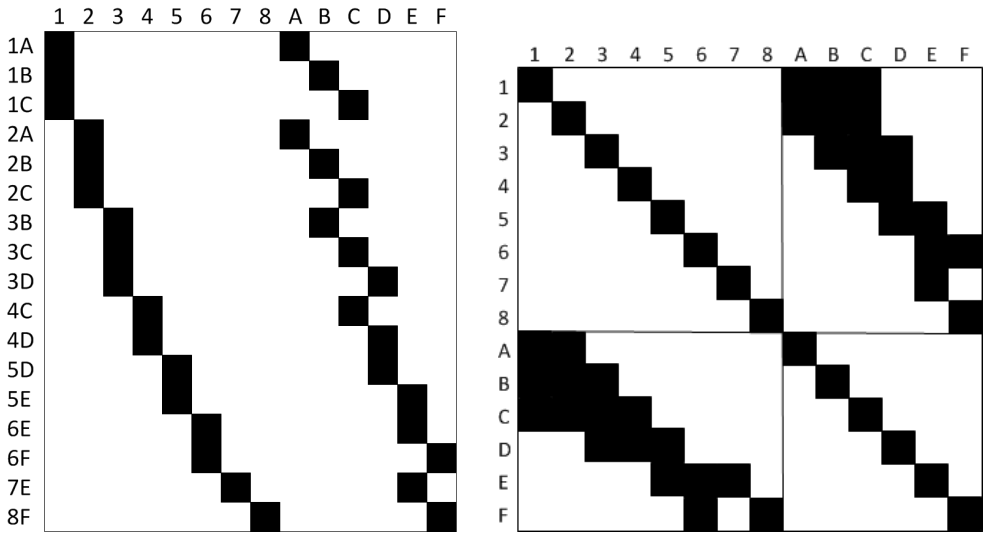
Given a linear system, $Mx = b$, the system can be partitioned as shown below.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} c \\ d \end{bmatrix} \tag{6.7}$$

In Equation 6.7, A, B, C, and D are block matrices. If A is invertible, we can form a reduced linear system with z as the unknown by arranging the equation.

$$(D - CA^{-1}B^T)z = d - CA^{-1}c \tag{6.8}$$

Here, $(D - CA^{-1}B^T)$ is the Schur complement of A. To efficiently compute the Schur complement, A should be easily invertible. The reduced linear system has a matrix with the same dimensions as D, which can be significantly smaller than the original matrix M. Once z has been solved, the solution for y can be obtained with a trivial back-substitution, thus solving the entire original problem.



(a) Jacobian J (b) $J^T J$

Figure 6.1: (a) The Jacobian matrix for a small bundle adjustment problem. On the horizontal axis, 1–8 correspond to point parameters and A–F correspond to camera parameters. The vertical axis refers to observations in the images. The derivatives of the point parameters (left) and the camera parameters (right) are partitioned horizontally to form a bipartite matrix. The point parameter blocks are sorted so that they align vertically. (b) The square matrix $J^T J$. The matrix is composed of four block submatrices, which can be used to from the Schur complement. In addition, the submatrices can be computed on-the-fly using the partitions of the bipartite Jacobian matrix.

6.0.1.2 Implicit Schur Complement

To review, we need to solve $J^T Jx = b$, where J is the Jacobian matrix. Due to the structure of J for bundle adjustment problems, we are able to avoid explicitly computing the Schur Complement. The structure of J is shown in Figure 6.1(a). Notice that the blocks in the left side of the matrix are arranged so that they stack vertically. The left and right partitions of J combine to form a bipartite matrix.

$$J = \begin{bmatrix} E & F \end{bmatrix} \tag{6.9}$$

It can be shown that given the structure of J as a bipartite matrix, $J^T J$ (shown in Figure 6.1(b)) can be decomposed into four block matrices, which are themselves the result of

operations on E and F.

$$\begin{bmatrix} E^T E & E^T F \\ F^T E & F^T F \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} c \\ d \end{bmatrix} \tag{6.10}$$

Due to the parameter blocks in E being aligned vertically, $E^T E$ is an easily invertible submatrix. We can thus form a Schur Complement-based reduced linear system using the block matrices.

$$(F^T F - F^T E(E^T E)^{-1} E^T F)z = d - F^T E(E^T E)^{-1} c \tag{6.11}$$

$$Sz = r \tag{6.12}$$

$$\text{where } S = (F^T F - F^T E(E^T E)^{-1} E^T F) \tag{6.13}$$

This formulation is called an implicit Schur complement because $J^T J$ is never explicitly computed with a sparse matrix-sparse matrix multiply (SpMSpM) and stored. Instead, it is formed by operations on the E and F partitions of the original bipartite matrix. We can now solve a reduced linear system and avoid computing and storing $J^T J$. From Figure 6.1(b), it is apparent that the size of the reduced linear system (the bottom-right submatrix) is determined by the number of cameras in bundle adjustment, which is usually significantly less than the number of points. In a conjugate gradient solver, the main computational bottleneck is the SpMV operations involving S. With the implicit Schur complement, each SpMV with S is done using 5 successive SpMVs with F, E^T, $(E^T E)^{-1}$, E, and F^T.

Additional optimizations are possible when solving large linear systems found in iterative non-linear least squares algorithms. In some implementations, Jacobian values can be computed on-the-fly and never have to be stored. More detailed information on such optimizations and on bundle adjustment in general can be found in the work of Wu et al. [91].

6.0.2 GPU Bundle Adjustment

Structure from motion (SfM) is the procedure of recovering the 3D structure of a scene from a set of 2D images. Structure-from-Motion has increasingly grown in size over time, especially

with the availability of large-scale community photo collections. As the number of photos in such collections continues to grow into the hundreds of thousands or even millions, the scalability of bundle adjustment algorithms has become a critical issue. Fortunately, we have also seen advancements in computer hardware used for massive data processing, including multi-core processors, many-core processors, and compute clusters. A key to speeding up bundle adjustment is to leverage these new processors.

There are different possible approaches for utilizing multiple GPUs in bundle adjustment. One could parallelize individual stages in bundle adjustment (such as the linear solver and the evaluation of Jacobian values). In a basic multi-GPU implementation of PCG (Preconditioned Conjugate Gradient), the multiple GPUs would need to synchronize with each other 5 times per iteration. For example, when we run traditional bundle adjustment on the Venice dataset, we find that the first 10 Levenberg-Marquardt iterations require 879 iterations of conjugate gradient. This means a multi-GPU implementation of PCG would need to synchronize the GPUs $879 \times 5 = 4395$ times. Such frequent synchronization could lead to performance degradation. Furthermore, as the problem size increases and the size of the linear system in the normal equations increases, the theoretical upper bound on the convergence of PCG (the dimension of the matrix) also increases. In practice, however, convergence usually occurs faster than this.

Due to the nature of the bundle adjustment problem, which forms two distinct set of parameters, cameras and points, another approach to parallelization is interleaved bundle adjustment, also known as Resection-Intersection. This scheme alternates between keeping the points fixed and optimizing the cameras (resection) and keeping the cameras fixed and optimizing the points (intersection). Either points or cameras can be easily optimized in parallel while the other remains fixed. The drawback of this approach is a slow convergence rate and/or convergence to a non-global optimum.

An alternative approach to parallelizing bundle adjustment is to partition the full problem into subproblems. Such an approach exploits the fact that bundle adjustment problems involve physical objects in the real world, which allow the problems to be partitioned in 3D space. This approach has the following advantages and disadvantages.

Advantages:

1. There is no need to synchronize separate GPUs.
2. There is less data transfer between GPU memory and host memory and between the memories of different GPUs.
3. It exploits the innate parallelism in the problem domain of 3D reconstruction. Many reconstruction problems can be partitioned naturally due to the physical partitioning of the components in the 3D scene.
4. When GPUs are used, a single machine with a shared-memory system can overcome a limitation in memory bandwidth when processing multiple subproblems in parallel.

Disadvantages:

1. The final solution can be less accurate. The partitioned bundle adjustment might not converge or converge to a non-global optimum.
2. There can be an issue of load balancing.
3. Partitioning and distributing work introduces overhead.
4. The choice and quality of partitioning can affect the outcome of optimization.

6.1 Related Work

There has been various works attempting to improve the performance of bundle adjustment. Ni et al. were the first to use graph partitioning to subdivide the bundle adjustment problem into subproblems [59]. They optimize independent partitions in parallel and then spatially re-align the partitions and points. Agarwal et al. study bundle adjustment featuring tens of thousands of images [4]. They explore different preconditioners when using conjugate gradient to solve the normal equations. Wu et al. were the first to implement bundle adjustment on the GPU [91]. However, their work does not support multiple GPUs, does not support double-precision, and has a fixed bundle adjustment cost function. Hänsch et al. use Resection-Intersection to perform parallel bundle adjustment on a GPU, but with a noticeable loss in accuracy [28]. Lakemond et

al. use a similar method with an augmented triangulation step between resection and intersection [45]. For distributed bundle adjustment, Erikkson proposes using ADMM based on point consensus [19]. Ramamurthy et al. also perform distributed bundle adjustment using ADMM but do so based on the consensus of both points and cameras in different partitions [66]. Zhang et al. use ADMM based on camera consensus to target multiple CPUs in a distributed cluster [95]. Demmel et al. do distributed photometric bundle adjustment targeting multiple CPU nodes [18].

6.2 Methodology

Our goal is to develop a method for doing parallel bundle adjustment that scales well with multiple GPUs. We make the following contributions:

- We analyze the ground truth accuracy of different parallel bundle adjustment methods including Resection-Intersection and ADMM with point consensus.
- We introduce a novel method that partitions the scenes and alternates optimizing the partitions and boundary points. Our method has better ground truth accuracy than Resection-Intersection, while still obtaining large speedups.
- We run our implementation on multiple GPUs and obtain large speedups when compared to a traditional, serial bundle adjustment.

6.2.1 Algorithm

In our method, which we call the Fixed-Boundaries method, we partition the scene using a graph cut. A set of boundary points is formed that connect the partitions. To optimize the partitions in parallel, we hold the boundary points fixed while optimizing each partition in parallel. Then, we keep the rest of the scene fixed while optimizing the boundary points. We continuously alternate in this manner until a stop criterion is reached. One such criteria is reaching a user-specified number of iterations. Another possible criteria is when the change in the objective function from the last iteration goes below a certain threshold, which can signal convergence. For this case, we look at the value of the objective function for the full bundle adjustment problem. During the optimization of both the partitions and the boundary points, we opt to only take a single step

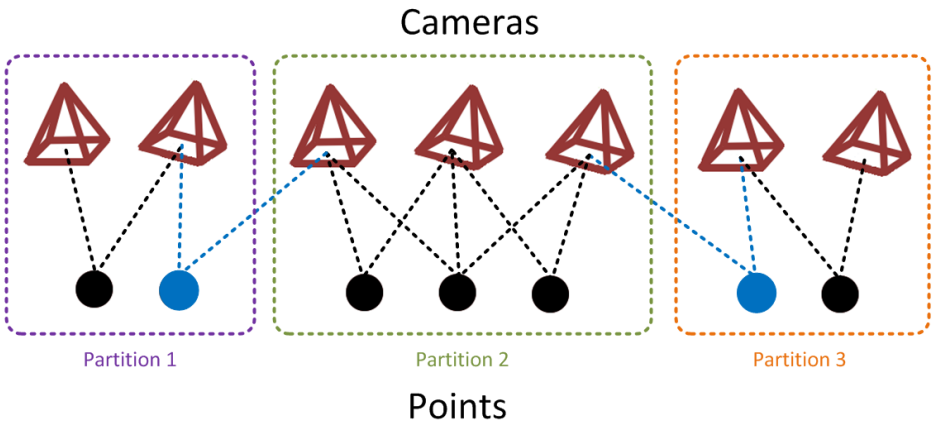


Figure 6.2: Example of partitioning a visibility graph. Each camera in the graph is connected to another camera via an edge if they view at least one common point. In this example, the partitioning is based on a min-cut and creates 3 partitions. Lines and points colored in blue indicate edges that have been cut. Points that are part of cut edges become boundary points.

of Levenberg-Marquardt. We do this to preserve stability, and we find that taking more than one step does not benefit the solution much in terms of accuracy. The method is summarized in Algorithm 4.

Algorithm 4 Partitioned Bundle Adjustment with Fixed Boundary Points

Use a minimum graph cut to divide the scene into N partitions and B boundary points
prev_f_value $\leftarrow 0$
for num_iterations **do**
  for all partitions **do** in parallel
    Take one successful optimization step with boundary points fixed
  for all boundary points **do** in parallel
    Take one successful optimization step with parameters in partitions fixed
  f_value $\leftarrow$ EVALUATEFUNCTION()
  if |f_value $-$ prev_f_value| $<$ function_tolerance **then break**
  prev_f_value $\leftarrow$ f_value

By holding the other parameters fixed while optimizing the boundary points, our method uses a technique similar to the intersection step in Resection-Intersection. Resection-Intersection is known to converge to a less optimal solution than that achieved by optimizing all parameters simultaneously. One cause is that during optimization in Resection-Intersection, a large number of parameters are fixed at any given time. Another cause is that Resection-Intersection fails to model the interactions between the set of points and the set of cameras, since one is always

fixed while the other is being optimized. In our method, many points and cameras can be optimized simultaenously when the boundary points are fixed. These boundary points can make up a relatively small fraction of the total points after using a partitioning based on a minimum graph cut.

One drawback of our implementation is its lack of support for cameras having shared intrinsics, since it assumes that every camera in the scene is different. This allows camera parameters in different partitions to be optimized separately. On the other hand, ADMM bundle adjustment based on camera consensus, as done by Zhang et al. [95], can support intrinsics shared among different cameras. However, the cameras would need to be duplicated in multiple partitions, which could introduce more overhead and nullify the performance benefits of parallel computation. Another insight about intrinsics is that for the case of a single camera viewing multiple parts of a scene, the intrinsics can be obtained from manufacturer specifications and do not need to be part of the bundle adjustment problem. Furthermore, in the case of randomly collected images, such as a collection of internet photos viewing a landmark scene, all cameras are generally assumed to be different.

6.2.2 Partitioning the Scene Graph

We opt to use a divide-and-conquer approach on the whole bundle adjustment problem, where each subproblem consists of a subset of the cameras and points in the original problem. To accomplish this, we partition the scene and bundle adjust each partition in parallel. This approach to parallelization can reduce the necessary communication between processors. The bundle adjustment of independent partitions can be scheduled as separate jobs that can be solved using multiple processors of potentially varying types, including GPUs.

First, we define the visibility graph for a scene. Each image forms a node in the graph and two images have a weighted edge between one another if there is a feature match across both images. Two images that have more feature matches between them will be connected by an edge of greater weight. Therefore, a subset of images will form a subgraph with higher edge weights if many features are tracked across this subset of images. As shown in Figure 6.2, a graph cut can be applied on the visibility graph to partition the problem into subproblems. When bundle adjusting subproblems separately, there is a potential problem of drift, where each partition

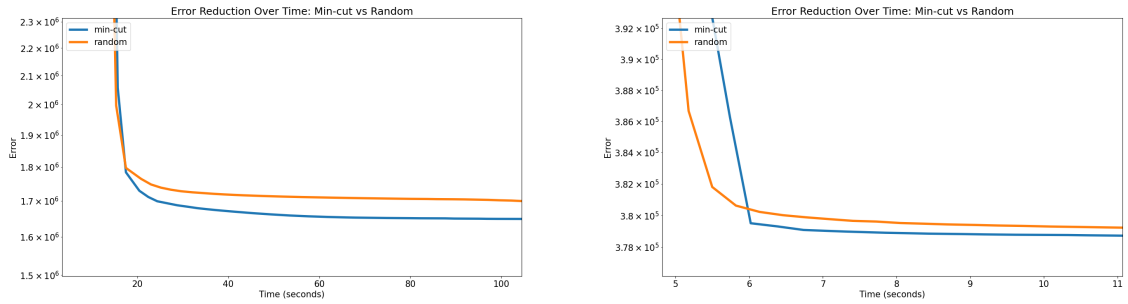
becomes locally optimized but is not optimized with respect to the entire problem. To deal with this issue, 3D points that are shared across partition boundaries (seen by images in different partitions) can be held constant during optimization. The boundary 3D points act as common "anchors" for the different subproblems.

For graph partitioning, we use the METIS library [39], which can partition based on a minimum cut of the graph. This leads to images that are the least connected to be placed in separate partitions. The assumption is that refinements done on one partition are less likely to depend on the refinements done on a less connected partition, which increases the independence of the subproblems. One issue of doing a straightforward min-cut partitioning of the visibility graph is the failure to take into account load balancing. Each image views a certain number of points, and the more points an image can see, the more computation is needed for that image when evaluating the cost function and the partial derivatives. Therefore, we opt to not only perform a min-cut purely based on the number of images viewing each point. Instead, we pass two lists to `METIS_PartGraphKway`. The first list consists of the *edge weights*, which are the weights of the edges in the visibility graph. The second list consists of *vertex weights*, which are the number of points seen by each image (where each image is a vertex in the visibility graph). By using both *edge weights* and *vertex weights* in the min-cut partitioning, we can balance between minimizing the connectivity between different partitions and improving the load balancing of future computation on the partitions. An example of partitioning a visibility graph based on a min-cut is shown in Figure 6.2. In this example, two edges are cut, and the number of cameras in each partition are 2, 3, and 2. The partitioning aims to cut as few edges as possible (and cut edges with less weight) while keeping the number of nodes (cameras) in each partition relatively even. As mentioned earlier in the discussion of vertex weights, the number of observations seen in each camera can also be used as additional weights during partitioning, though the number of observations in a partition often correlates with the number of cameras in the partition.

The goal of a min-cut partitioning is to keep the number of boundary points low. This has two advantages: (1) There are fewer boundary points to optimize after optimizing the partitions in parallel, thus saving runtime. (2) Having fewer boundary points requires fewer points to be

Table 6.1: Number of boundary points after partitioning

Dataset	# Cams	# Pts	# Obs	M-Cut # boundary points	Rand. # boundary points	M-Cut % total points	Rand. % total points
Venice	1778	993923	5001946	235036	936520	24%	94%
Ladybug	1723	156502	678718	27812	146026	18%	94%
Final	13682	4456117	28987644	1032552	4173407	23%	94%



(a) Venice (b) Ladybug

Figure 6.3: Error reduction over time using two different partitionings: min-cut and random. Both partitionings are tested on two different datasets: (a) Venice and (b) Ladybug.

fixed while the partitions are being optimized in parallel. This allows more points to optimize simultaneously *with* the cameras, thus leading to a larger reduction in error per iteration. In a sense, a min-cut partitioning provides the benefits of interleaving without suffering from slow convergence caused by fixing a large percentage of the parameters during optimization. We test the effectiveness of a min-cut partitioning by comparing it against a random partitioning of the images. We run these tests and all following tests on 4 NVIDIA V100 GPUs that are shared among 8 partitions. For the CPUs, we use Intel Xeon E5-2698 v4 CPUs running at 2.20 GHz.

Table 6.1 shows the number of boundary points on three different datasets that result from a min-cut partitioning and a random partitioning. The Ladybug dataset is derived from images captured at a regular rate using a Ladybug camera mounted on a moving vehicle, which increases the sparsity of the problem. As shown in the table, the number of boundary points that result from a random partitioning makes up a large percentage of the total number of points. By using a random partitioning, the implementation becomes close to Resection-Intersection bundle adjustment. Figure 6.3 shows the performance of parallel bundle adjustment using the two

different partitionings on the datasets Venice and Ladybug. The random partitioning performs worse in the same manner as Resection-Intersection bundle adjustment, where convergence is less optimal due to a large number of parameters being fixed during optimization.

6.2.3 Framework

We create our implementation on top of the Ceres Solver library. Ceres [3] is an open-source C++ library for modeling and solving large-scale optimization problems. Ceres can be used to solve non-linear least squares problems with bounds constraints as well as general unconstrained optimization problems. This library is commonly used to solve bundle adjustment problems for 3D reconstruction.

Ceres contains different algorithms for solving the large, sparse linear systems that occur in bundle adjustment problems. These algorithms include Cholesky Factorization, a direct solver for medium to small systems, and conjugate gradient, an iterative solver for large systems. Variations of the solvers can take advantage of the structure of the Jacobian matrix, including those that leverage the Schur complement trick and use an implicit Schur complement representation to create a reduced linear system. As mentioned previously, the size of the Schur complement in a bundle adjustment problem is usually bounded by the number of cameras in the problem.

We use Ceres' existing functionality to set parameters constant during optimization. To improve the performance of our approach, we modify the solver to allow a Ceres problem to pause optimization (after taking a successful step of optimizing each partition) and resume (after taking a successful step of optimizing the boundary points) with updated parameters. This removes the overhead of setting up a new problem and any necessary preprocessing that is needed to begin minimization.

6.2.4 GPU Acceleration

Generally, the most time-intensive operation in solving bundle adjustment problems is solving a linear system during the computation of each Levenberg-Marquardt step. When it comes to large-scale bundle adjustment problems, the linear solver of choice is preconditioned conjugate gradient. This method is feasible because the matrix in the linear system is quite sparse. Ceres has support for different preconditioners. We opt to use the block Jacobi preconditioner, as it

is the default in Ceres and often yields the fastest solution. We parallelize the linear algebra operations used in conjugate gradient (norms, dot products, etc) on a GPU using CUBLAS and CUSPARSE. As discussed previously, performing a partitioned bundle adjustment allows us to avoid GPU-to-GPU communication, as each GPU solves its own subproblem independently.

The second most time-intensive operation in bundle adjustment is the computation of partial derivatives needed to form the Jacobian matrix. The number of residuals in our optimization problem (the total number of tracked features) determines the number of partial derivatives that need to be computed. For large-scale problems, especially those with high-resolution images and an aggressive feature detector, the number of residuals can be quite high.

6.2.4.1 Automatic Differentiation on the GPU

Ceres uses automatic differentiation as a flexible approach for calculating the Jacobian values of user-defined cost functions [3]. Auto-differentiation is performed through the use of dual numbers. As explained in the documentation for Ceres, one way to understand dual numbers is to use complex numbers as an analogy. Both can be considered extensions of real numbers. The two differ in that complex numbers augment real numbers with an imaginary variable i where $i^2 = -1$, while dual numbers add an infinitesimal variable ϵ where $\epsilon^2 = 0$. As such, a dual number has both a real component and an infinitesimal component. In the following example, we can see how dual numbers can be used to compute derivatives of a function without having to work with symbolic mathematical expressions:

$$f(x) = x^2$$

Evaluate the function at $x = 5$:

$$f(5 + \epsilon) = (5 + \epsilon)^2 = 25 + 10\epsilon + \epsilon^2 = 25 + 10\epsilon$$

Observe that the value of the function at $x = 5$ is 25 and the coefficient of ϵ is 10. The latter is the correct value of the first-order derivative of the function at $x = 5$. Since we often deal with multivariate functions in optimization, we need to generalize dual numbers to Jets. A Jet consists of a single real part and a N-dimensional infinitesimal part. The following defines the datatypes within a `Jet` template.

```
template<int N> struct Jet {
  double a;
  Eigen::Matrix<double, 1, N> v;
};
```

Next, we look at the Ceres approach to defining custom cost functions. To implement the function from the previous example, we would write:

```
struct Function {
  template <typename T> bool operator()(const T* x,
                                        T* residual) const {
    residual[0] = x[0] * x[0];
    return true;
  }
}
```

The function's `operator()` would be passed a `Jet<1>` at runtime, and after evaluation its `v` data member would contain the coefficient of the infinitesimal, which is the value of a partial derivative at `x`. For the previous example to compile, run, and produce the correct result, the `Jet` data structure would need to have an overloaded operator:

```
template<int N> Jet<N> operator*(const Jet<N>& f,
                                 const Jet<N>& g) {
  return Jet<N>(f.a * g.a, f.a * g.v + f.v * g.a);
}
```

To support different cost functions, not only for bundle adjustment but also for a variety of non-linear least squares problems, many operators for `Jet` need to be overloaded to support a range of mathematical expressions. To make this implementation of auto-differentiation work on the GPU, we need to append CUDA keywords to the relevant functions and overloaded operators:

```
struct Function {
  template <typename T>
  __HOST__ __DEVICE__
  bool operator()(const T* x, T* residual) const
  {...}
}

template<int N>
__HOST__ __DEVICE__
Jet<N> operator*(const Jet<N>& f, const Jet<N>& g)
{...}
```

CUDA currently supports nearly all the same mathematical operations as the C++ standard. While this direct mapping from host to device code may not be the most efficient approach (for example, it does not consider optimizing GPU memory access via coalescing), it keeps the code readable. However, additional code is still needed to transfer the cost functions between the host machine and the GPU. Regardless, with this approach, each cost function can compute its partial derivative in a separate GPU thread, leading to a high amount of parallelism. Currently, we do not support mixing different cost functions in a single problem when computing derivatives in parallel on the GPU. Doing so would require multiple kernel launches, as each different type of cost function would need its own templated kernel.

As far as we know, we are the first to implement bundle adjustment in C++/CUDA with support for user-defined cost functions on the GPU. Supporting user-defined functions that can be auto-differentiated on the GPU enables further research in accelerating optimization problems with GPUs. Unlike other GPU-based implementations such as that of Wu et al. [91], our implementation is not limited to a set of fixed bundle adjustment cost functions. Instead, our use of auto-differentiation allows users to leverage GPUs regardless of the camera model they employ for their problems. More generally, auto-differentiation on the GPU enables accelerated Jacobian computation for a variety of non-linear least squares problems.

6.2.4.2 GPU Data Layout

Algorithm 5 shows simplified pseudo-code for taking a single step of Levenberg-Marquardt. The main routines that run on the GPU include solving the linear system using conjugate gra-

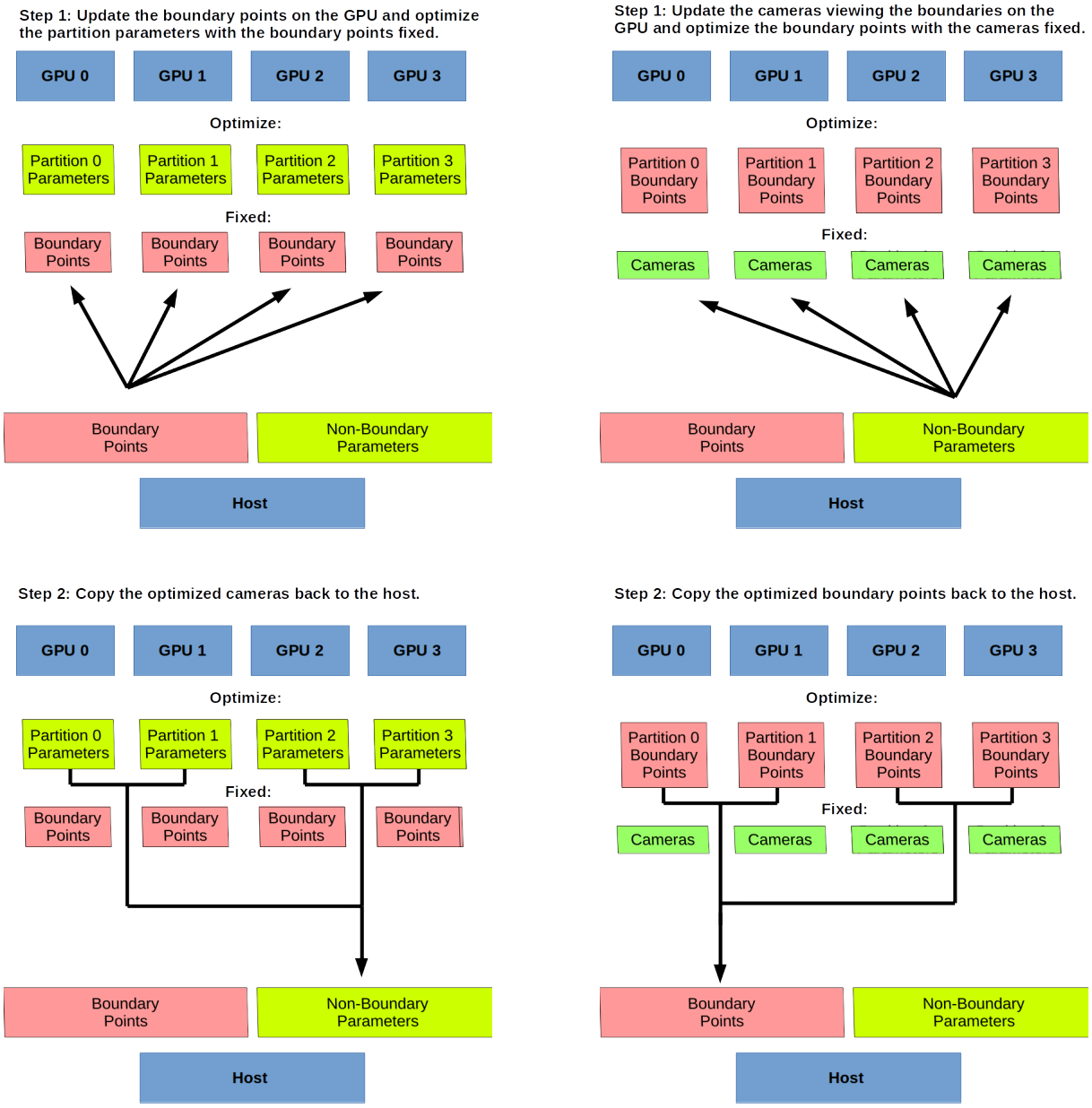
Algorithm 5 A Single Step of Levenberg-Marquardt on the GPU

Input: A vector function f,
A measurement vector $\mathbf{y}$,
A vector of fixed parameters $\mathbf{c}$,
An initial parameters estimate $\boldsymbol{\beta_0}$
Output: A parameter vector $\boldsymbol{\beta}^+$ minimizing $||\mathbf{y} - f(\mathbf{c}, \boldsymbol{\beta})||^2$.

procedure LEVENBERGMARQUARDTGPU
  $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta_0}$
  $\mu \leftarrow$ INITIALIZETRUSTREGIONSIZE()
  step_taken $\leftarrow$ **false**
  repeat
    $\mathbf{r} \leftarrow$ EVALUATEFUNCTIONGPU(f, $\mathbf{y}$, $\mathbf{c}$, $\boldsymbol{\beta}$)
    $\mathbf{J} \leftarrow$ AUTODIFFERENTIATEGPU(f, $\mathbf{y}$, $\boldsymbol{\beta}$)
    $\lambda \leftarrow$ COMPUTEDAMPINGPARAMETER(μ)
    Solve with GPU Conjugate Gradient $(\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})\delta_{\boldsymbol{\beta}} = \mathbf{J}^T\mathbf{r}$
    $\boldsymbol{\beta}_{\mathbf{new}} \leftarrow \boldsymbol{\beta} + \delta_{\boldsymbol{\beta}}$
    $\mathbf{r}_{\mathbf{new}} \leftarrow$ EVALUATEFUNCTIONGPU(f, $\mathbf{y}$, $\mathbf{c}$, $\boldsymbol{\beta}_{\mathbf{new}}$)
    $\boldsymbol{\Delta}_r \leftarrow ||\mathbf{r}||^2 - ||\mathbf{r}_{\mathbf{new}}||^2$
    if $\boldsymbol{\Delta}_r > 0$ **then**
      $\boldsymbol{\beta}^+ \leftarrow \boldsymbol{\beta}_{\mathbf{new}}$
      step_taken $\leftarrow$ **true**
    else
      $\mu \leftarrow$ SHRINKTRUSTREGIONSIZE(μ)
  until step_taken $==$ **true**

dient (this involves multiple steps of SpMV and dot products), evaluating the cost function, and using auto-differentiation on the cost function to compute the Jacobian values. Note that the algorithm expects certain parameters, **c**, to be fixed during optimization. When optimizing parameters in each partition, these fixed parameters are the boundary points. When optimizing the boundary points, these fixed parameters are the cameras that view the boundary points.

For our GPU computations, our data layout is straightforward but not highly optimized. For evaluating the cost functors to compute Jacobian values, the functors are stored simply as an array-of-structures. Future work would explore the more optimal structure-of-arrays. To store a computed Jacobian on the GPU, the Jacobian matrices are written to a format similar to Block Compressed Sparse Row (BCSR). The format is modified to support different sized rectangular blocks instead of equally sized square blocks. Storing in a block format can simplify the computation of the Jacobian values, since these values follow a block pattern within the



(a) Optimizing a subproblem in a partition

(b) Optimizing the boundary points

Figure 6.4: In our method, we alternate optimizing the partition parameters with the boundary points fixed and the boundary points with the cameras fixed. In the first stage (a), the boundary points on each GPU must be updated before they can be fixed during optimization. After taking a single step of Levenberg-Marquardt, the optimized camera parameters are copied back to the host. In the second stage (b), these camera parameters are sent to each GPU to update the parameters that will be fixed during optimization of the boundary points.

matrix. For example if there are 9 parameters per camera and 3 parameters per point, then the Jacobian will consist of multiple 2x9 and 2x3 blocks (with one partial derivative per parameter for the x-coordinate in the image and one partial derivative per parameter for the y-coordinate). For GPU-accelerated linear algebra operations, such as solving linear systems and performing SpMV in conjugate gradient, the matrix must be converted to Compressed Sparse Row format (CSR) before it can be passed to CUSPARSE or CUBLAS routines. In our method, the matrix structure remains the same across multiple optimization iterations because the partitioning does not change during optimization. Therefore, once the data structure for a matrix has been set up on the GPU, we only need to periodically copy matrix values after each iteration of the optimizer. We do not need to copy matrix structure data, such as the row indices for a CSR matrix.

For multiple GPUs, the divide-and-conquer approach applies to two sets of partitions: (1) the partitioning of the visibility graph that creates multiple bundle adjustment subproblems and (2) the assignment of boundary points to different GPUs. In the latter case, each boundary point can be optimized independently with all cameras fixed. In the first case, each partition has a large set of point parameters and, to a lesser extent, camera parameters that can be optimized independently without ever needing to communicate these values across partitions. Thus, for many points and cameras in a subproblem, each partition can store its own copies of Jacobian values and cost functors. After each partition has been optimized with a single step (Algorithm 5), the boundary points are optimized. Before this can proceed, updated camera parameters that are connected to the boundary points must be copied back to host memory and then sent to the memories of the GPUs. The number of these updated camera parameters depend on the connectivity of the visibility graph, but are generally small compared to the total number of point parameters. After the boundary points are optimized with the camera parameters fixed, they must be copied back to host memory and then sent to the memories of different GPUs based on the assignment of boundary points to GPUs. Doing so provides updated boundary points that are fixed during the optimization of the independent subproblems. For this data transfer, the number of boundary points determines the amount of data that needs to be copied.

During the optimization of either the subproblems or the boundary points, no GPU-to-GPU

communication is necessary. The flow of data between the GPUs and the host during the alternated optimization steps are shown in Figure 6.4. Note that the assignment of boundary points to each GPU in the second stage is not the same as that used in the first stage. Since each boundary point can be optimized independently, they can be equally divided among the GPUs in the second stage for perfect load balancing. In contrast, during the first stage, each GPU needs boundary points that are visible in its partition. For the assignment of camera parameters in the second stage, each GPU only needs parameters for cameras viewing its boundary points.

6.3 Results

6.3.1 Accuracy Results

To measure accuracy after bundle adjustment, ground truth can be used if available. In this case, ground truth would provide the correct positions of the cameras and points and an operation such as Iterative Closest Point (ICP) [8] can be used to align the correct point cloud with the bundle adjusted result to measure a distance between the two. However, in many cases, there is a lack of ground truth information, especially for new reconstructions done from random collections of images. After all, if ground truth is available, there would be no need to do the reconstruction in the first place. As a result, reprojection error is typically used in the absence of ground truth. The issue is that reprojection error can theoretically be low even if the result is of poor quality. Whether or not the final result matches the true answer, reprojection error will be low simply if the cameras, points, and feature tracks all agree with each other. This can happen, for example, if the input data is degenerate and converges to an "optimal" but incorrect solution. Local minima can also be problematic, as optimization can reach a low but not the lowest error. It would be fruitful to see if this actually occurs in practice, or if reprojection error does indeed have a strong correlation with ground truth.

Multiple works have explored different approaches for parallel bundle adjustment, though few of them verify the methods with ground truth data, opting instead to only use reprojection error. In the following sections, we discuss some known methods for parallel bundle adjustment, along with their advantages and disadvantages. Next, we test these methods, as well as our own novel method, on two synthetically generated datasets with artificial noise added to them. We

observe the accuracy of different methods when compared to ground truth solutions.

6.3.1.1 Resection-Intersection

Resection-Intersection, also known as interleaving, performs bundle adjustment by alternating fixing the points and optimizing the cameras (resection) and fixing the cameras and optimizing the points (intersection). This method can treat each camera and point independently and therefore optimize them in parallel. In terms of accuracy, there are some notable disadvantages. As noted by Hartley and Zisserman, the method can be competitive with a full bundle adjustment when the scene is small and highly interconnected [30]. However, in less favorable conditions, accuracy can suffer and performance gains from parallelization may be overshadowed by slow convergence rates. Another potential downside of Resection-Intersection involves the communication costs in a distributed system. Depending on the connectivity of cameras and points in the scene, each point may depend on a large percent of the cameras in the scene and vice versa. Combined with the large number of iterations required for this method, significant amounts of global communication may be needed to broadcast the cameras and points after each iteration of optimization.

Hänsch et al. study the parallelization of Resection-Intersection on a GPU [28]. As expected, they find that the obtained reprojection-error is often not as low as that of a full bundle adjustment. Though they do not compare their results with any ground truth data, they are able to obtain superior error reduction per time spent during the early iterations. Lakemond et al. augment their Resection-Intersection method with a triangulation step prior to the Intersection stage [45]. To preserve stability, they also track the change in reprojection error involving each point after the resection step and only optimize the points that exhibit a large enough change in their contributed reprojection errors. On some datasets, they achieve a final reprojection error that is comparable with that of a normal bundle adjustment. However, they do not test their method on any ground truth data. The authors for both works acknowledge that Resection-Intersection can lead to slower convergence and less accurate results due to many parameters being fixed during each step of optimization. In contrast, we try to keep more parameters free during optimization by only fixing the boundary points.

6.3.1.2 ADMM

An algorithm that has become popular in recent times for performing parallel partitioned bundle adjustment is the Alternating Directions Method of Multipliers (ADMM). Since the bundle adjustment problem is not a convex function, certain assumptions have to made to ensure convergence in ADMM. As noted by Zhang et al. [95] and Mayer [52], a condition for convergence requires that the points not be too close to the cameras, which is a reasonable assumption for SfM problems.

One advantage of ADMM is that partitions of the scene can be optimized to near completion in parallel during one ADMM iteration. During this parallel optimization, no partition needs to communicate with any other partition. Only the gathering and scattering of the consensus variables at the end of an ADMM iteration require global communication, which can make the overall algorithm more efficient in distributed environments. ADMM has some disadvantages. As stated by Boyd et al. [10], the algorithm is known to converge slowly to a highly accurate solution when one is desired. Furthermore, the convergence behavior is highly sensitive to the value of the penalty parameter ρ, including both its initial value and an increase/decrease factor that can be applied during each ADMM iteration. There are also issues that can arise when formulating the bundle adjustment problem as an ADMM problem. For example, if the new primal function is not formulated as a least-squares problem, then one cannot use an existing least-squares solver to minimize the function. Another possibility is that the new formulation cannot be minimized as efficiently using the same least squares solver that is used for a full bundle adjustment function. For example, Ramamurthy et al. [66] use a Gauss-Newton solver instead of Levenberg-Marquardt to solve the new primal function in their ADMM formulation. Issues can also arise relating to overhead and scalability during parallel computation. For example, the added penalty terms in the new ADMM formulation can lead to an increase in necessary computation. Other issues relating to overhead will be discussed shortly.

6.3.1.2.1 Point Consensus

When formulating bundle adjustment as an ADMM problem, one can opt to create a consensus problem based on points or cameras. In the case of point consensus, the same point can appear in multiple partitions, and a consensus value for the point is updated during each iteration.

For camera consensus, a similar formulation applies for cameras instead of points. Boyd et al. provide a general overview for formulating ADMM problems [10]. Equations 6.14, 6.15, and 6.16 formulate an ADMM bundle adjustment problem based on point consensus.

$$(C)^{t+1}, (X^l)^{t+1} = \underset{C,X^l}{\operatorname{argmin}} \left(f_l((C)^t, (X^l)^t) + \frac{\rho}{2} ||(X^l)^t - (X)^t + (\tilde{X}^l)^t||_2^2 \right) \tag{6.14}$$

$$(X_j)^{t+1} = \frac{1}{L} \sum_{l=1}^{L} (X_j^l)^{t+1} \tag{6.15}$$

$$(\tilde{X}_j^l)^{t+1} = (\tilde{X}_j^l)^t + (X_j^l)^{t+1} - (X_j)^{t+1} \tag{6.16}$$

In these equations, C represents a vector of cameras, X_j represents the jth point in a vector of points, and l is used to enumerate a partition out of L total partitions. The ADMM iteration number is specified with t. Equation 6.14 gives the function that needs to be optimized for a single partition, which can be done independently from other partitions. The function $f_l((C)^t, (X^l)^t)$ is the objective function of the bundle adjustment subproblem containing C, the subset of cameras in partition l, and X^l, the subset of points in partition l. X encapsulates the consensus values of these points, and $\tilde{X}^l$ are the dual variables for these points. These two terms are updated using the following two equations. For points shared across different partitions, Equation 6.15 globally averages each of these points to form a consensus. Equation 6.16 updates the dual variables in accordance with the theory of ADMM to drive the consensus points towards convergence. The variable ρ is the penalty parameter, which can have a large impact on the convergence behavior of the algorithm.

Eriksson et al. perform ADMM bundle adjustment using point consensus to target distributed systems [19]. However, they only test their distributed implementation on small datasets and do not test on ground truth data. Mayer follows Eriksson et al. and performs point consensus bundle adjustment on larger datasets but only on two of them [52]. To avoid the dilemma of choosing the penalty parameter ρ and having to update ρ as the optimization progresses, the author uses the covariance matrix of the boundary points as weights to the penalty term instead

of ρ. He achieves speedup on a multi-core CPU but, like Eriksson et al., he does not test his implementation on ground truth data.

6.3.1.2.2 Camera Consensus

Zhang et al. are the first to test ADMM bundle adjustment on very-large scale scenes [95]. Unlike previous methods, they formulate bundle adjustment as a camera consensus problem. The motivation is that in most reconstructions, the points greatly outnumber the cameras. When using point consensus, there is potentially a large number of boundary points that need to reach consensus, which can lead to slower convergence, as the numerous points have to be averaged. Furthermore, these consensus points would have to be globally communicated after each ADMM iteration, which could lead to performance degradation, particularly in distributed environments where communication costs are high.

The motivation for camera consensus is sensible for some datasets but may have drawbacks for others. Performing camera consensus requires the same cameras to appear in multiple partitions. For highly disjoint scenes, there would be little overlap of cameras in different partitions. However, for scenes where the visibility graph has high connectivity, there could potentially be large overlap of cameras in different partitions. In the worst case, with N cameras and M points in the scene and P partitions, each partition would have N cameras. Even if the partitioning is able to achieve M/P points per partition, the large number of cameras per partition could limit performance gains. The main runtime bottleneck for Levenberg-Marquardt is solving the linear system formed by the normal equations. Solving this system, as described earlier, can be done efficiently by first creating the reduced Schur-complement system, due to the Jacobian matrix being formed by two distinct sets of parameters: points and cameras. The size of the reduced Schur-complement system depends on the number of cameras, since the points typically outnumber the cameras. By having overlapping cameras in each partition, dividing the scene into P partitions does not necessarily decrease the size of the reduced Schur-complement system in each partition by a proportional factor. Therefore, camera consensus can have scalability issues on certain types of scenes. Zhang et al. run their distributed bundle adjustment based on camera consensus on multiple large scenes. However, they do not compare the scalability and runtime performance of their implementation with that of a full, serial bundle adjustment. They also do

not verify the accuracy of their bundle adjustment on ground truth data.

ADMM bundle adjustment can also be formulated as a consensus problem of both cameras and points, as done by Ramamurthy et al. [66]. They test their implementation on synthetic ground truth data but only add noise to the points. Their real datasets have 30 or fewer cameras and their largest synthetic dataset has 2000 cameras.

6.3.1.3 Fixed Boundary Points

Our method of parallelizing bundle adjustment by fixing boundary points has some advantages. First, compared to Resection-Intersection, we converge to a better solution because within each partition, the cameras and points are optimizing together. We also have less global communication than Resection-Intersection, because only the boundary points and the cameras viewing them need to be globally communicated. Compared to ADMM, our method does not require overlapping cameras or points in different partitions. Overlapping cameras, as in the case of ADMM based on camera consensus, can be inefficient because even after partitioning, each subproblem can still have a large number of cameras. Unlike ADMM, our method also does not require setting and managing a penalty parameter ρ, which can lead to better convergence rates.

Our method also has some disadvantages. As we'll see in some tests, the method is better suited for certain types of scenes and may not converge to the optimal solution for other scenes. Our method works best when the scene is highly connected, with numerous shared points between different cameras and having varying viewpoints of each point. Compared with ADMM, our implementation may require more global communication, as we take only one step of optimization during each iteration.

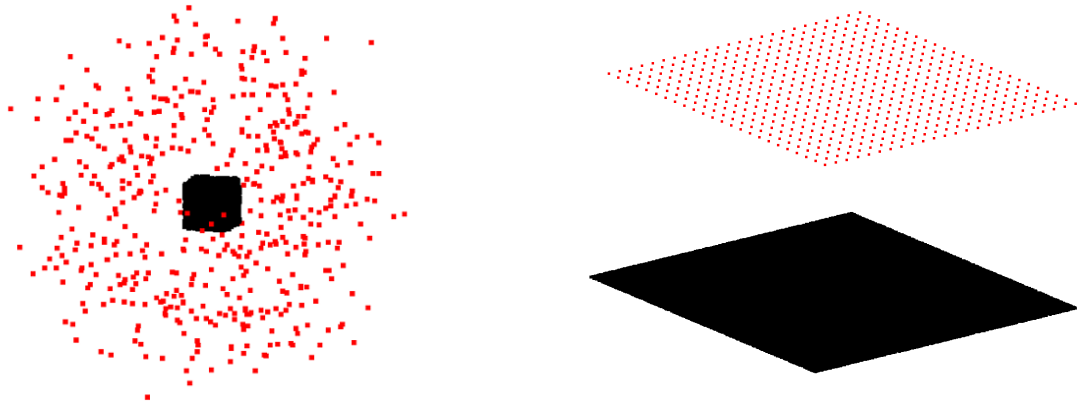
6.3.1.4 Ground Truth Results

To test different parallel bundle adjustment methods, we generate synthetic data to use as ground truth. We create two synthetic scenes, keeping them small, since we are interested in accuracy and not performance for these tests. We add noise to the scenes, but we keep the amount of noise low to ensure that bundle adjustment can converge to the correct, ground truth solution. The first dataset $sphere$ is a cube of points surrounded by cameras in all directions. The set of camera positions roughly form a sphere with random perturbations. There are 500 cameras and 10000 points. Each point is visible in 10 randomly selected cameras, and each camera is made

to view at least roughly 300 points. With this configuration, the visibility graph of the scene has high connectivity, and each point is viewed by multiple cameras from varying vantage points. To add noise to this scene, we add a maximum rotation error of 0.1 to each of the three angle-axis rotation parameters in the camera model. We also add maximum camera translation errors of 5 and maximum point position errors of 5.

In our second synthetic scene, $grid$, we arrange 576 cameras in a 24x24 grid all looking vertically downwards at a random set of points that roughly form a plane beneath the grid of cameras. This dataset simulates a scene derived from aerial footage. In the camera grid, each camera is spaced 8 units apart, and each camera is 125 units above the plane of points. The criteria for a point being viewed by a camera is as follows: if the camera's position is projected onto the plane of points, the points within a radius of 20 from the projected position are visible in the camera. As a result, a camera in the middle of the grid shares points with 47 other cameras, and every point is visible in at least 12 cameras. Cameras in the visibility graph are connected to other nearby cameras that view the same points. Cameras that are far apart are not directly connected. Therefore, this synthetic scene is much less well-connected than the previous one. For artificial noise, since it's easy to converge to a non-global mininmum in this scene, we add a maximum rotation error of just 0.001, a maximum translation error of 0.1, and a maximum point position error of 0.1.

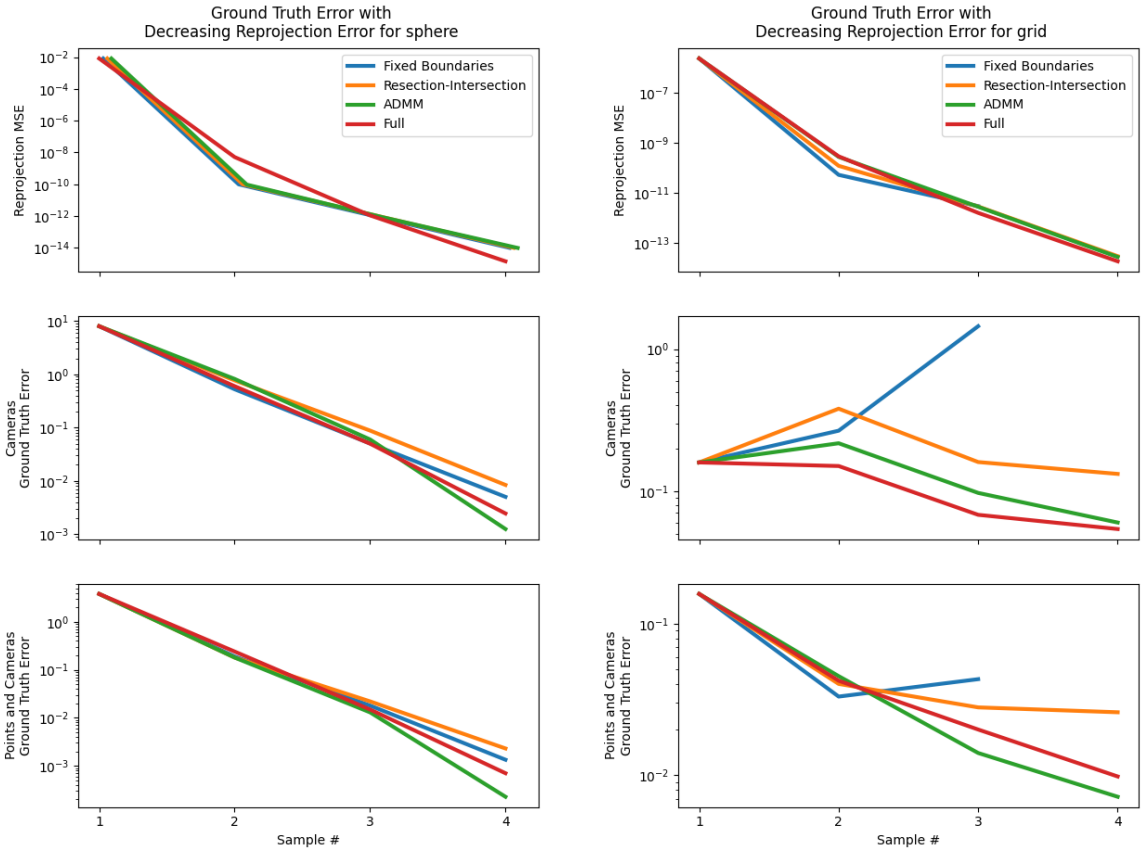
To measure ground truth error after performing bundle adjustment, we must take into account the global translation, rotation, or scaling difference that can exist between the result of bundle adjustment and ground truth. We use CloudCompare's ICP algorithm (that also accounts for scaling) to align the bundle-adjusted result and ground truth, before measuring the average distance between the point clouds containing the camera and point positions. To test the accuracy of different parallel bundle adjustment methods, we observe the ground truth error as the mean squared reprojection error decreases. We sample the progress of each method at four different intervals, which includes the initial cost prior to the start of optimization. In one graph, we show the error reduction for only camera positions, and in the other, we show the error reduction for both camera and point positions. Generally, in bundle adjustment problems, the final accuracy of the cameras is more important than the accuracy of the points.



(a) $sphere$ (b) $grid$

Figure 6.5: The ground truth cameras and points for synthetic datasets $sphere$ and $grid$. Cameras are in red and points are in black.

For the fixed boundaries method and ADMM, we first need to partition the problem. We use METIS to divide the scene into four partitions using a minimum graph cut. Figure 6.6a shows the results of a full bundle adjustment, Resection-Intersection, our fixed boundaries method, and ADMM on the $sphere$ dataset. Though we cannot always compare the results of the different methods directly, due to the reprojection errors at the sample intervals being different, we note that an effective bundle adjustment method should lower ground truth error as it lowers reprojection error. We see that the full non-parallel bundle adjustment successfully converges to ground truth as the reprojection error decreases. The Resection-Intersection method converges to a solution with a bit more error, ADMM converges to an even better ground truth-solution than the full bundle adjustment (given similiar reprojection errors), and our method is somewhere in between. This confirms previous works showing less accurate results from Resection-Intersection. In our implementation of Resection-Intersection, we only perform one iteration of minimization for each of the alternated steps, as we find that additional iterations do not lead to more accuracy. Our fixed boundaries method, while not the most accurate, is able to lower ground truth error alongside reprojection error. This confirms that our method does progress towards a lower ground truth error. Our method optimizes the problem in an alternating fashion and gets a similar benefit of being parallelizable like Resection-Intersection but converges to a

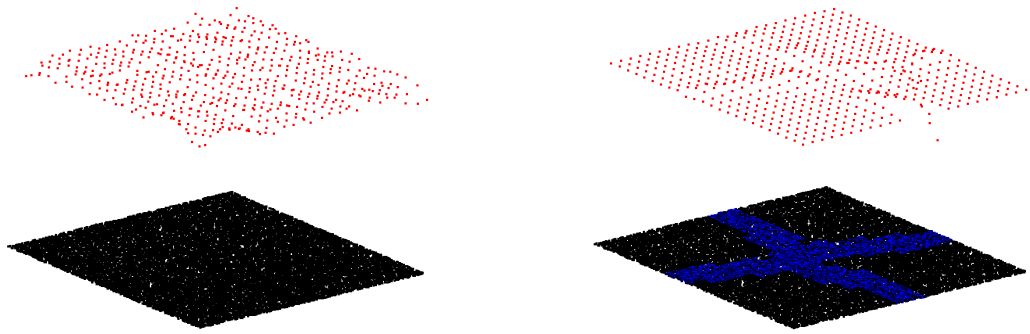


(a) $sphere$ (b) $grid$

Figure 6.6: A comparison of three parallel methods–fixed boundaries, Resection-Intersection, and ADMM–and one serial method–a normal full bundle adjustment. The graph tracks the decrease or increase of ground truth error (for cameras only and cameras/points) as reprojection error decreases for (a) the $sphere$ dataset and (b) the $grid$ dataset.

solution with lower error.

The results for $grid$ are shown in Figure 6.6b. For this scene, the full bundle adjustment, Resection-Intersection, and ADMM all reduce the ground truth error along with the reprojection error. For Resection-Intersection, we also experiment with taking more than one minimization iteration during the resection stage (optimizing the cameras with the points fixed) to see the effect on stability. As shown in Figure 6.7a, after just one resection step with multiple minimization iterations, the camera positions move vertically towards or away from the plane of points. Because a camera's movement in these directions hardly changes the reprojection error, the optimization of this scene is ill-conditioned and can easily converge to a non-global mini-



(a) Incorrect solution for Resection-Intersection. (b) Incorrect solution for the fixed boundaries method.

Figure 6.7: In the $grid$ dataset, the camera positions (shown in red) should correctly form a rectangular plane. For this dataset, (a) shows convergence to an unoptimal solution when taking too many resection steps in Resection-Intersection, and (b) shows our fixed boundaries method converging to an unoptimal solution, which is largely caused by the divergent optimization of cameras viewing the boundary points (shown in blue).

mum. Therefore, similar to the test on the previous scene, we use the result of taking only one step during resection and one step during intersection.

Looking at the result of our fixed boundaries method, the optimization converges to a non-global minimum. The reprojection error becomes stuck in a local minimum (which is the reason its last sample is not shown in the graph), and this increases the ground truth error during optimization. In Figure 6.7b, we see certain cameras having incorrect vertical movement towards or away from the plane of points. These cameras are viewing many boundary points (shown in blue), which are fixed during the optimization of the cameras. Compared to the previous scene, few other cameras view the same boundary points as the problematic cameras. For the ones that do, there lacks a wide enough baseline between them to help constrain the problem and prevent the problematic cameras from moving towards a non-optimal position. This suggests that our method is best suited for scenes that are more highly connected and that have varying viewpoints for each point. For more disjointed datasets such as $grid$, we would suggest Resection-Intersection or ADMM as more applicable parallel bundle adjustment methods.

6.3.1.5 Discussion

We have tested different parallel bundle adjustment methods on ground truth data to measure their effectiveness in reaching a low ground truth error. As expected, we find that Resection-

Intersection does not converge to as optimal a solution as other methods. We find that for scenes with certain properties, our method can reach a low ground truth error that correlates with low reprojection error similar to a full bundle adjustment. It has the same advantage of being parallelizable like Resection-Intersection, but achieves lower error. The drawback is that it fails for certain scene configurations, namely those that are ill-conditioned. Such problems can easily optimize towards a non-global minimum. Most importantly, we verify that achieving a low reprojection error with our method also achieves low ground truth error, which is useful for gauging the method's effectiveness for real datasets. The method works better with highly connected scenes with a variety of viewpoints for each point. We believe that dense photo-collections of an object, such as tourist photos of a popular landmark gathered from the internet, would fit this criteria. In the future, improved processing power and storage would enable the collection of more images, the collection of larger images, and the ability to detect and match more features per image. This will lead to more dense and well-connected reconstruction problems.

6.3.2 Performance Results

We test our implementation on a single-node machine with multiple GPUs and multiple CPU cores. Multi-threading is implemented using POSIX threads. The CPU used for all serial and parallel tests is the Intel Xeon E5-2630 v3 running at 2.40 GHz, and the GPUs are 4 NVIDIA V100s. We obtain our real datasets from the Bundle Adjustment in the Large (BAL) project managed by the University of Washington GRAIL Lab. This project explores reconstruction from large-scale online image collections [4]. The provided bundle adjustment problems were created by running photos through a reconstruction software and outputting intermediate, un-optimized reconstructions to disk at certain time intervals. The Ladybug dataset comes from images captured at a regular rate using a Ladybug camera mounted on a moving vehicle. Photographs that contributed to the Venice dataset were downloaded en masse from Flickr.com.

6.3.2.1 Scaling

First, we test the scalability of the partitioned bundle adjuster by tracking runtime while changing the number of partitions from 2 to 8. We run these tests using 4 GPUs on the Venice dataset and the Ladybug dataset. Figure 6.8(a) and Figure 6.9(a) shows the runtime scaling on these

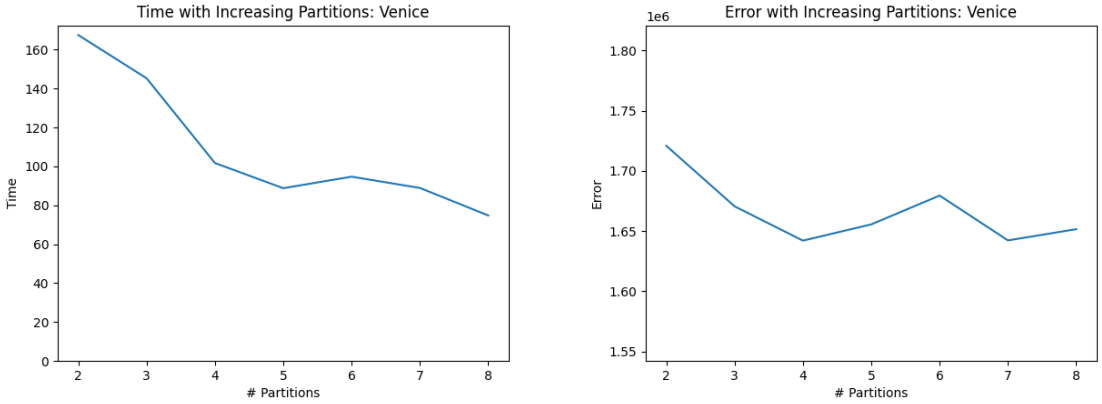
datasets running from 2 to 8 partitions/cores. Our goal is to measure the strong scaling of the parallel implementation. However, it is difficult to ensure that each implementation with a different number of partitions is given the same amount of work to do, as changing the number of partitions essentially changes the optimization problem. To approximate this, we simply have all implementations run 30 alternated iterations. For the Venice dataset, the GPUs appear to become saturated once each GPU is assigned at least one partition (when using 4 or more partitions). Using 8 partitions improves performance marginally, and after 4 partitions, using a number of partitions that is not a multiple of the number of GPUs decreases performance slightly. For the Ladybug dataset, the runtime appears to stop improving significantly after 3 partitions. The Ladybug is a smaller dataset and converges rapidly to a good solution, which causes the algorithm to run out of work quickly. For larger datasets, our implementation is able to saturate the GPUs with useful work but appears to be sensitive to the load balancing on the GPUs. Currently, once we assign a partition to a GPU, the assignment is fixed throughout the optimization. Future work would look into a dynamic scheduling of partitions onto available GPUs to improve multi-GPU load balancing.

Figure 6.8(b) and Figure 6.9(b) show the amounts of error achieved when using different numbers of partitions. Because each optimization problem is different due to a different partitioning, the error achieved after taking a set number of iterations are not the same. However, the differences are much less than an order of magnitude.

6.3.2.2 Real Datasets Performance

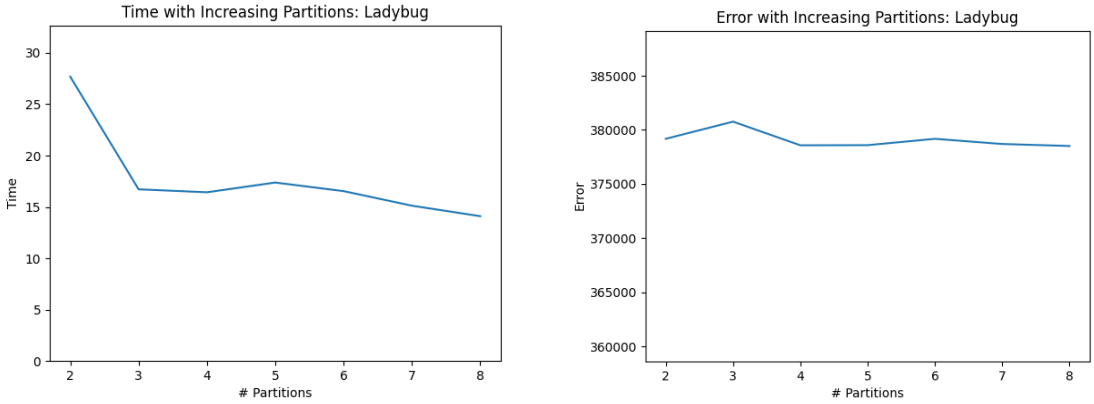
Next, we run some full performance tests by comparing our parallel GPU-accelerated implementation using the modified Ceres Solver with the unmodified Ceres Solver running in serial. We plot the total amount of reprojection error versus runtime. For all these tests, we use 8 partitions. The partitioned implementation is able to obtain good speedup on the different datasets. The serial and parallel implementations are able to converge to similar amounts of overall error.

The synthetic dataset features 2304 cameras laid out in a regular 48x48 grid. The cameras look downward at a plane containing randomly scattered points. Each camera views a minimum of 400 points. Like the *grid* scene in the accuracy tests, neighboring cameras in the 48x48 grid have a small amount of overlap in terms of point visibility. We add a large amount of artificial



(a) Time (b) Error

Figure 6.8: The (a) runtime and (b) obtained error for the Venice dataset as the number of partitions increase.



(a) Time (b) Error

Figure 6.9: The (a) runtime and (b) obtained error for the Ladybug dataset as the number of partitions increase.

noise to this synthetic dataset, which likely prevents the optimization from converging to a good solution. However, the excessive noise is added to give bundle adjustment more work to do, enabling its use as a runtime benchmark. The sizes of all datasets and the achieved speedups are summarized in Table 6.2.

From the graphs, we can see that parallel bundle adjustment has competitive accuracy with the full serial bundle adjustment, while obtaining large speedups. In the graphs, a speedup is defined as the serial runtime divided by the parallel runtime when the optimization results for

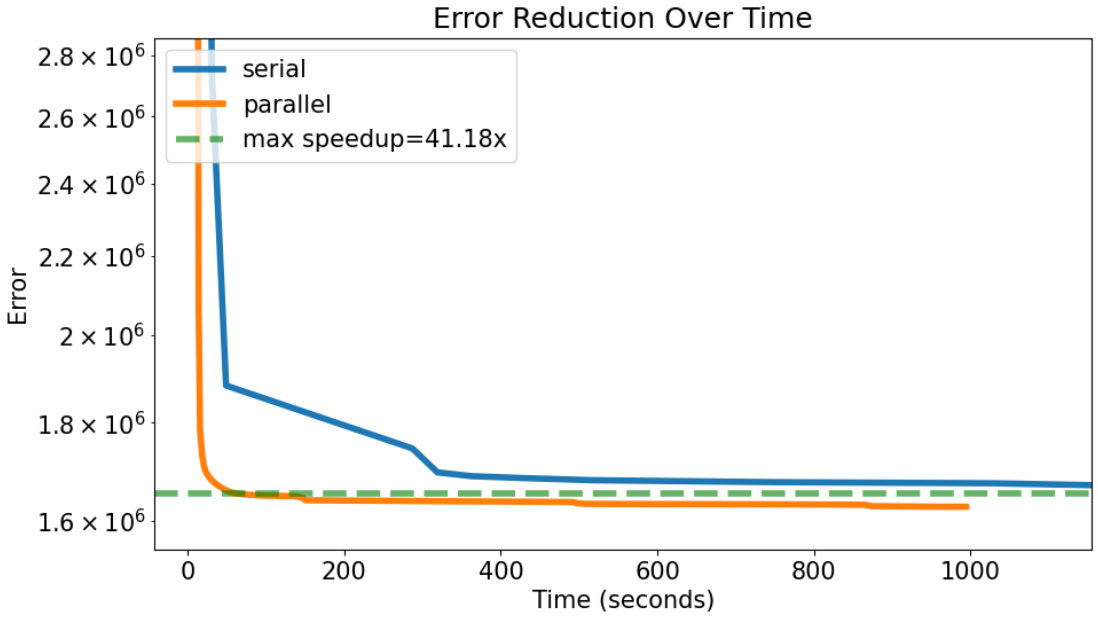


Figure 6.10: Venice dataset: error reduction over time and max speedup.

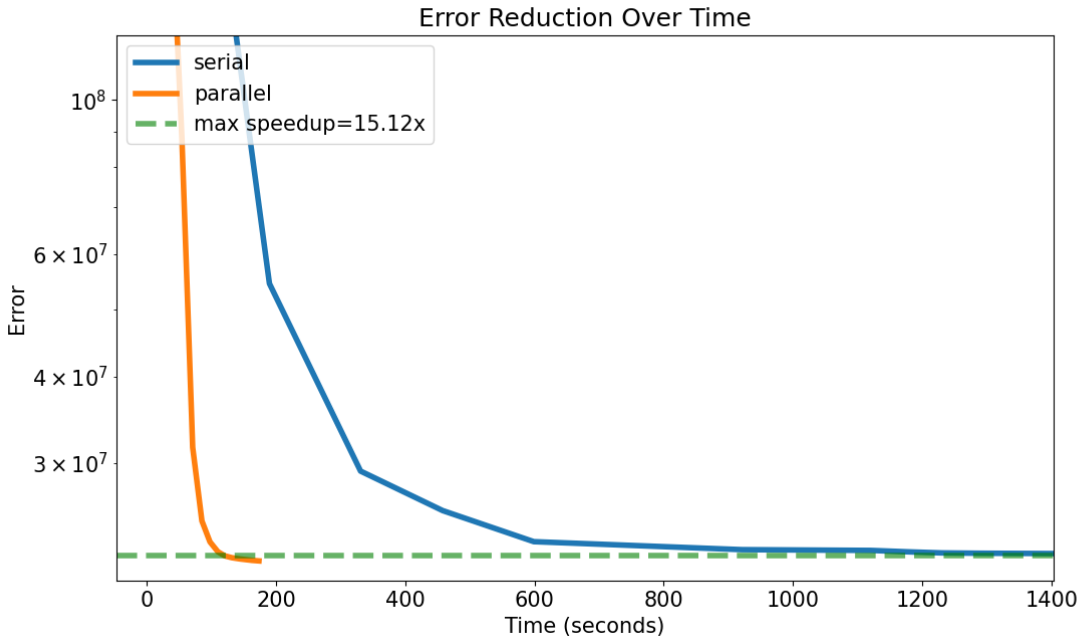


Figure 6.11: Final dataset: error reduction over time and max speedup.

both implementations have reached a similar amount of error. The max speedup is then the greatest among all the possible speedups that occur where both implementations have reached

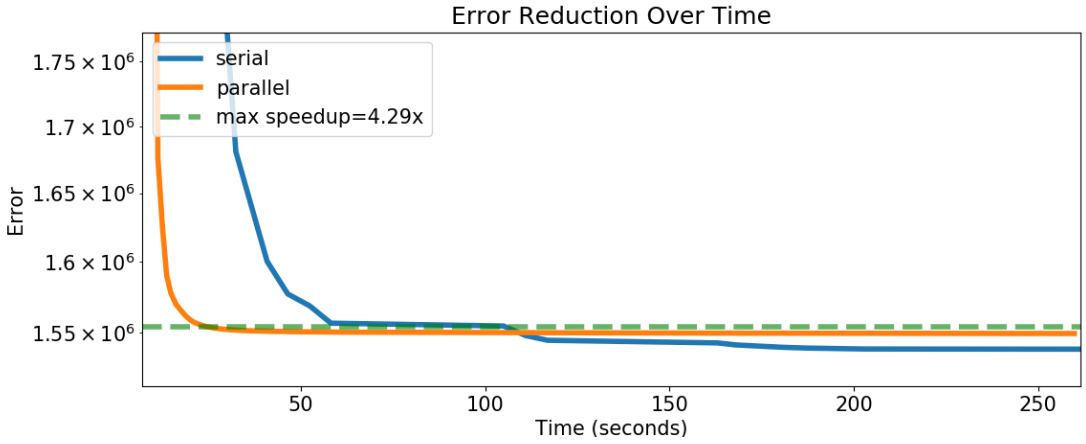


Figure 6.12: Rome09 dataset: error reduction over time and max speedup.

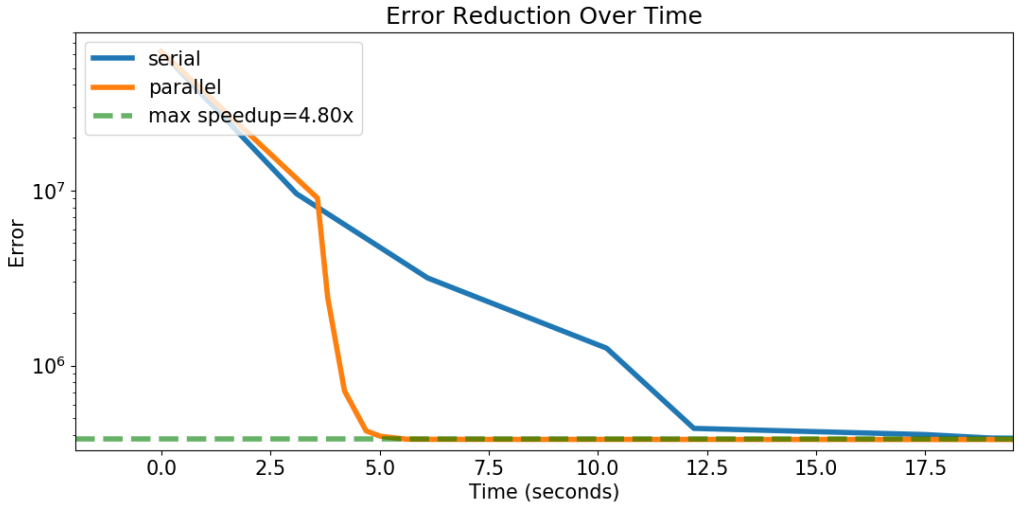


Figure 6.13: Ladybug dataset: error reduction over time and max speedup.

a similar amount of error. In the graphs, this is the largest horizontal "gap" between the two curves and is denoted by a dashed horizontal line.

In some cases, the parallel method does not reach the same accuracy as the serial method, and in other cases, it reaches a lower error. For the Rome09 dataset, the parallel method's worse accuracy is likely explained by the inability to create good partitions. This dataset features views of a single building and does not contain multiple disjoint components. After partitioning, a majority of points become boundary points, which means many points need to be held constant during optimization. For this dataset, the parallel method becomes closer to Resection-Intersection, which is known to optimize to a less accurate solution than a full bundle adjust-

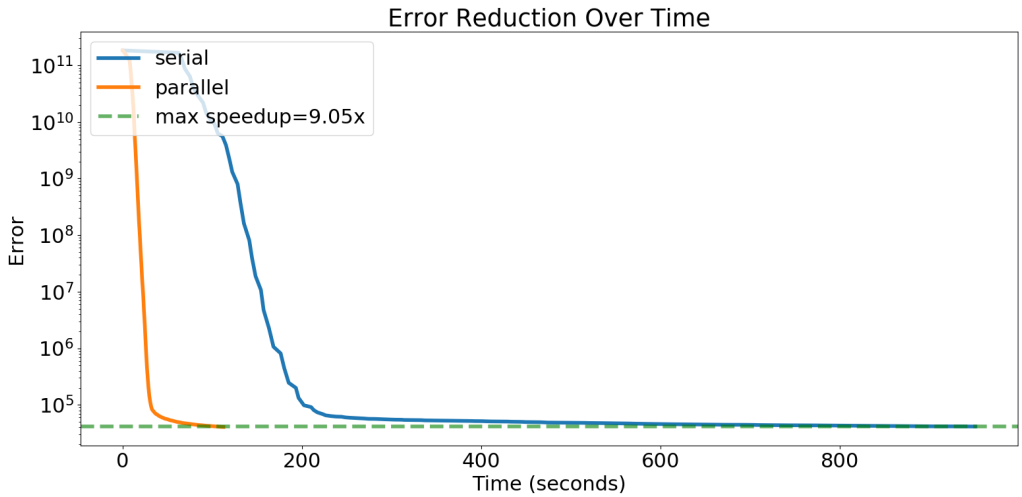


Figure 6.14: Error reduction over time and max speedup of a synthetic dataset with 2304 cameras, 921600 points, and 1783967 observations.

ment.

Table 6.2: Maximum Parallel Speedup

Dataset	Cameras	Points	Observations	Max Speedup
Venice	1778	993923	5001946	41X
Final	13682	4456117	28987644	15X
Rome09	6983	49983	3078434	4X
Ladybug	1723	156502	678718	4.8X
Synthetic-48x48	2304	921600	1783967	9X

The maximum parallel speedup obtained for different datasets.

6.4 Conclusion and Future Work

We learn some important lessons in our work on parallel bundle adjustment. First, a highly disjoint partitioning (e.g. a min-cut based partitioning) can lead to large speedups and reduced communication across processors, but such a partitioning can converge to a less accurate or even unusable solution. Another determining factor is the structure of the scene, as demonstrated by the *sphere* and *grid* datasets. In the case of *grid*, the scene is ill-conditioned and does not form a well-constrained problem. When using a disjoint partitioning, the scene structure can cause optimization to fall easily into a local minimum. In cases where the problem has better constraints

(e.g. *sphere*), a good solution can be reached despite a disjoint partitioning. Using a less disjoint partitioning can affect runtime performance due to the overhead of overlapping parameters but is less likely to converge to a local minimum. For example, Resection-Intersection fixes many parameters during each step of optimization, making convergence slow, but still ends up with a feasible solution for the *grid* dataset. Similarly, the partitionings used in ADMM implementations typically have more overlap among the partitions. But even for scenes such as *grid*, such implementations can converge to highly accurate solutions, though possibly at the cost of runtime. These insights can be taken into consideration when choosing how to partition a bundle adjustment problem, where there is a trade-off between more parallelism and the likelihood of converging to a correct solution. Information about the scene should also be considered. In the case where the input data is abundant and the camera coverage of the scene is thorough and complete, a more disjoint partitioning can be used without the risk of converging to a poor solution. In the future, we expect that the amount of data collected for reconstruction will continue to increase, leading to scenes that are more complete and with less uncertainty. In this scenario, more disjoint partitions can be a good choice. For scene structures that are not well-conditioned, future work can explore how to mitigate accuracy problems while still allowing for a disjoint, highly parallel implementation. Some of these directions can build on top of existing work, such as developing GPU implementations of ADMM based on point consensus or camera consensus. Another approach could use multiple partitionings and alternate the boundary variables that are selected to be fixed in each iteration. Such an approach can prevent optimization from taking steps that lead to instability. When it comes to evaluating approaches and measuring accuracy, we would also like to create and test new bundle adjustment problems using the benchmarks of Knapitsch et al. [40], as these are large-scale and include ground truth.

Even with the trade-offs in mind, our parallel bundle adjustment does appear to perform well in terms of speed and accuracy on real datasets. We verify this by first testing the accuracy of different parallel bundle adjustment methods on ground truth data. We confirm that for our method, a lower reprojection error correlates with a lower ground truth error. In our performance results on large-scale real datasets, we are able to obtain reprojection errors that are competitive with or even better than those of a full serial bundle adjustment. Our two-level parallelization of

(1) dividing the scene into partitions and (2) using GPUs to accelerate the optimization of each partition, enables us to obtain large speedups. We believe that we have developed an efficient method to leverage multiple GPUs for large-scale bundle adjustment.

Chapter 7

Conclusion

This chapter concludes this dissertation and provides some insights on future work involving 3D reconstruction on GPUs. Reconstruction uses a large amount of data which is ever-increasing, so naturally, many stages of the reconstruction pipeline are memory bound problems. In terms of GPU hardware, reconstruction can clearly benefit from improved memory bandwidth on GPUs. This includes the bandwidth of accessing DRAM as well as the bandwidth between host and device memory. A unified memory between host and device memory is a powerful tool as long as the synchronization between the two is automatic and high-performing. Faster GPU-to-GPU communication will be another useful tool in not only shared memory systems but also distributed memory systems. Distributed clusters are a key platform to handle the growing scale of reconstruction problems. Such systems will also allow for reconstruction to happen offline as a cloud service.

Though hardware will continue to try to keep pace with the growing demands of reconstruction problems, software poses an even greater challenge. While images are useful and common, they often have ambiguities that lead to accuracy problems. As shown in previous chapters, uncertainty and ambuigity in a dataset can have a large impact on the quality of the final result. Other sensors have a key role to play in the future of reconstruction. Different types of sensors working together is important for removing the ambiguities of purely image-based reconstruction. The use of multiple sensors is especially prevalent in growing fields such as self-driving cars. Flexible programming frameworks are key to integrating the data from these sensors in a meaningful way. The challenge is even greater when the goal is to leverage devices such as

GPUs to keep runtime tractable.

In the previous chapter, we discussed auto-differentiation as a tool for enabling customizability in a reconstruction pipeline. A user is able to define custom cost functions for optimization, which can then be auto-differentiated. Such tools are valuable because reconstruction, along with many other computer vision problems, typically involve some type of global optimization. A user should be able to integrate data from a variety of sensors without the burden of having to modify an existing programming framework. Such flexibility will make collecting and leveraging more data an attractive option. A next step is to support auto-differentiation on one or more GPUs seamlessly. In our work, we support only a single type of cost function running on the GPU at a time. An important goal is to support multiple types of cost functions and to use CUDA programming constructs such as streams to enable them to be assigned efficiently to GPUs. Doing so enables the ability to feed many sources of data into a GPU-accelerated reconstruction application. Work scheduling should be automatic without sacrificing too much performance. Data movement between host and device or among multiple GPUs should be hidden from the user but run fast, and on distributed systems, communication should be minimized.

With accelerated auto-differentiation, research and development for sensor fusion becomes easier. New algorithms and problem formulations can be quickly prototyped and tested at a large scale. For example, in our bundle adjustment implementation, we could test new formulations where we assign different weights to the boundary points, and run these implementations directly on multiple GPUs. Likewise, one could choose to increase the weights of sensor data deemed more reliable for bundle adjustment or similar optimization problems. Custom cost functions also enable different formulations of ADMM with different partitioning schemes to be implemented and tested more quickly. Programming frameworks for sensor fusion will be key to supporting the variety of reconstruction problems that will inevitably crop up. But with such software and massively parallel GPUs, reconstruction will become much more accesible and open up new applications.

REFERENCES

[1] A.E. Abdel-Hakim and A.A. Farag. CSIFT: A SIFT Descriptor with Color Invariant Characteristics. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1978–1983, 2006. https://dx.doi.org/10.1109/CVPR.2006.95

[2] Sameer Agarwal, Manmohan Krishna Chandraker, Fredrik Kahl, David Kriegman, and Serge Belongie. Practical Global Optimization for Multiview Geometry. In *Proceedings of the 9th European Conference on Computer Vision - Volume Part I*, ECCV'06, page 592605. Springer-Verlag, Berlin, Heidelberg, 2006. https://dx.doi.org/10.1007/11744023_46

[3] Sameer Agarwal, Keir Mierle, et al. Ceres Solver. http://ceres-solver.org, 2020.

[4] Sameer Agarwal, Noah Snavely, Steven M. Seitz, and Richard Szeliski. Bundle Adjustment in the Large. In *Proceedings of the 11th European Conference on Computer Vision: Part II*, ECCV'10, page 2942. Springer-Verlag, Berlin, Heidelberg, 2010.

[5] Hernn Badino, Daniel F. Huber, and Takeo Kanade. Integrating LIDAR into Stereo for Fast and Improved Disparity Computation. In *International Conference on 3D Imaging, Modeling, Processing, Visualization and Transmission* (edited by Michael Goesele, Yasuyuki Matsushita, Ryusuke Sagawa, and Ruigang Yang), pages 405–412. IEEE Computer Society, 2011.

[6] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded Up Robust Features. In *Proceedings of the 9th European Conference on Computer Vision - Volume Part I*, ECCV'06, pages 404–417. Springer-Verlag, Berlin, Heidelberg, 2006. https://dx.doi.org/10.1007/11744023_32

[7] Nathan Bell and Jared Hoberock. Thrust: A Productivity-Oriented Library for CUDA.

In *GPU Computing Gems*, volume 2 (edited by Wen-mei W. Hwu), chapter 26, pages 359–371. Morgan Kaufmann, October 2012.

[8] P.J. Besl and Neil D. McKay. A Method for Registration of 3-D Shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992. https://dx.doi.org/10.1109/34.121791

[9] Mårten Björkman, Niklas Bergström, and Danica Kragic. Detecting, Segmenting and Tracking Unknown Objects Using Multi-label MRF Inference. *Computer Vision and Image Understanding*, 118:111–127, January 2014. https://dx.doi.org/10.1016/j.cviu.2013.10.007

[10] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and Trends in Machine Learning*, 3:1–122, 01 2011. https://dx.doi.org/10.1561/2200000016

[11] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[12] Martin Byröd, Klas Josephson, and Kalle Åström. Fast Optimal Three View Triangulation. In *Proceedings of the 8th Asian Conference on Computer Vision*, ACCV'07, pages 549–559. Springer-Verlag, Berlin, Heidelberg, 2007.

[13] F. Calakli, A. O. Ulusoy, M. I. Restrepo, G. Taubin, and J. L. Mundy. High Resolution Surface Reconstruction from Multi-view Aerial Imagery. In *Proceedings of the 2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization & Transmission*, 3DIMPVT '12, pages 25–32. IEEE Computer Society, Washington, DC, USA, 2012. https://dx.doi.org/10.1109/3DIMPVT.2012.54

[14] J. Y. Chang, H. Park, I. K. Park, K. M. Lee, and S. U. Lee. GPU-friendly Multi-view Stereo Reconstruction Using Surfel Representation and Graph Cuts. *Computer Vision and Image Understanding*, 115(5):620–634, May 2011. https://dx.doi.org/10.1016/j.cviu.2010.11.017

[15] Changchang Wu. VisualSFM: A Visual Structure from Motion System. http://homes.cs.washington.edu/~ccwu/vsfm/, 2011.

[16] William G. Cochran. *Sampling Techniques, 3rd Edition*. John Wiley, 1977.

[17] Zhijun Dai, Yihong Wu, Fengjun Zhang, and Hongan Wang. A Novel Fast Method for L_∞ Problems in Multiview Geometry. In *Proceedings of the 12th European conference on Computer Vision – Volume Part V*, ECCV'12, pages 116–129. Springer-Verlag, Berlin, Heidelberg, 2012. https://dx.doi.org/10.1007/978-3-642-33715-4_9

[18] Nikolaus Demmel, Maolin Gao, Emanuel Laude, Tao Wu, and Daniel Cremers. Distributed Photometric Bundle Adjustment. In *2020 International Conference on 3D Vision (3DV)*, pages 140–149, 2020. https://dx.doi.org/10.1109/3DV50981.2020.00024

[19] Anders Eriksson, John Bastian, Tat-Jun Chin, and Mats Isaksson. A Consensus-Based Framework for Distributed Bundle Adjustment. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1754–1762, 2016. https://dx.doi.org/10.1109/CVPR.2016.194

[20] C.H. Esteban and F. Schmitt. Silhouette and stereo fusion for 3D object modeling. In *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.*, pages 46–53, 2003. https://dx.doi.org/10.1109/IM.2003.1240231

[21] Martin A. Fischler and Robert C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Commun. ACM*, 24(6):381395, jun 1981. https://dx.doi.org/10.1145/358669.358692

[22] Andrew W. Fitzgibbon, Geoff Cross, and Andrew Zisserman. Automatic 3D Model Construction for Turn-Table Sequences. In *Proc. of the European Workshop on 3D Structure from Multiple Images of Large-Scale Environments*, pages 155–170. Springer-Verlag, London, UK, 1998.

[23] Yasutaka Furukawa, Brian Curless, Steven M. Seitz, and Richard Szeliski. Towards Internet-Scale Multi-View Stereo. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1434–1441, 2010. https://dx.doi.org/10.1109/CVPR.2010.5539802

[24] Yasutaka Furukawa and Jean Ponce. Accurate, Dense, and Robust Multiview Stereopsis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(8):1362–1376, August 2010. https://dx.doi.org/10.1109/TPAMI.2009.161

[25] David Gallup, Jan-Michael Frahm, Philippos Mordohai, Qingxiong Yang, and Marc Pollefeys. Real-Time Plane-Sweeping Stereo with Multiple Sweeping Directions. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007. https://dx.doi.org/10.1109/CVPR.2007.383245

[26] Michael Goesele, Noah Snavely, Brian Curless, Hugues Hoppe, and Steven M. Seitz. Multi-View Stereo for Community Photo Collections. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007. https://dx.doi.org/10.1109/ICCV.2007.4408933

[27] J-Y Guillemaut and A. Hilton. Joint Multi-Layer Segmentation and Reconstruction for Free-Viewpoint Video Applications. *International Journal of Computer Vision*, 93(1):73–100, May 2011. https://dx.doi.org/10.1007/s11263-010-0413-z

[28] Ronny Hänsch, Igor Drude, and Olaf Hellwich. Modern Methods of Bundle Adjustment on the GPU. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, III-3:43–50, 2016. https://dx.doi.org/10.5194/isprs-annals-III-3-43-2016

[29] R. I. Hartley and P. Sturm. Triangulation. pages 146–157. Elsevier Science Inc., New York, NY, USA, 1997.

[30] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. 2nd edition. Cambridge University Press, 2004.

[31] Richard Hartley and Fredrik Kahl. Optimal Algorithms in Multiview Geometry. In *Proceedings of the 8th Asian conference on Computer vision – Volume Part I*, ACCV'07, pages 13–34. Springer-Verlag, Berlin, Heidelberg, 2007.

[32] Carlos Hernandez, George Vogiatzis, and Roberto Cipolla. Probabilistic Visibility for Multi-View Stereo. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007. https://dx.doi.org/10.1109/CVPR.2007.383193

[33] Mauricio Hess-Flores, Mark A. Duchaineau, and Kenneth I. Joy. Sequential Reconstruction Segment-wise Feature Track and Structure Updating Based on Parallax Paths. In *Proc. of the 11th Asian Conference on Computer Vision - Volume Part III*, ACCV'12, pages 636–649. Springer-Verlag, Berlin, Heidelberg, 2013. https://dx.doi.org/10.1007/978-3-642-37431-9_49

[34] Christian Hne, Lionel Heng, Gim Hee Lee, Alexey Sizov, and Marc Pollefeys. Real-Time Direct Dense Matching on Fisheye Images Using Plane-Sweeping Stereo. In *2014 2nd International Conference on 3D Vision*, volume 1, pages 57–64, 2014. https://dx.doi.org/10.1109/3DV.2014.77

[35] Forrest N. Iandola, David Sheffield, Michael J. Anderson, Phitchaya Mangpo Phothilimthana, and Kurt Keutzer. Communication-minimizing 2D convolution in GPU registers. In *2013 IEEE International Conference on Image Processing*, pages 2116–2120, 2013. https://dx.doi.org/10.1109/ICIP.2013.6738436

[36] Google Inc. Google Earth. https://www.google.com/earth/.

[37] John Isidoro and Stan Sclaroff. Stochastic Refinement of the Visual Hull to Satisfy Photometric and Silhouette Consistency Constraints. In *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*, ICCV '03, page 1335. IEEE Computer Society, USA, 2003.

[38] K. Kanatani, Y. Sugaya, and H. Niitsuma. Triangulation from Two Views Revisited: Hartley-Sturm vs. Optimal Correction. In *Proceedings of the British Machine Vision Conference*, pages 18.1–18.10. BMVA Press, 2008.

[39] George Karypis and Vipin Kumar. Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing 20(1), 359-392, 01 1999. https://dx.doi.org/10.1137/S1064827595287997

[40] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction. Association for Computing Machinery, New York, NY, USA, Jul 2017. https://dx.doi.org/10.1145/3072959.3073599

[41] K. Kolev, M. Klodt, T. Brox, and D. Cremers. Continuous Global Optimization in Multiview 3D Reconstruction. *International Journal of Computer Vision*, 84(1):80–96, 2009. https://dx.doi.org/10.1007/s11263-009-0233-1

[42] K. Kolev, T. Pock, and D. Cremers. Anisotropic Minimal Surfaces Integrating Photoconsistency and Normal Information for Multiview Stereo. In *Proceedings of the 11th European Conference on Computer Vision Conference on Computer Vision: Part III*, ECCV'10, pages 538–551. Springer-Verlag, Berlin, Heidelberg, 2010.

[43] Ilya Kostrikov, Esther Horbert, and Bastian Leibe. Probabilistic Labeling Cost for High-Accuracy Multi-view Reconstruction. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1534–1541, 2014. https://dx.doi.org/10.1109/CVPR.2014.199

[44] K. N. Kutulakos and S. M. Seitz. A Theory of Shape by Space Carving. *International Journal of Computer Vision*, 38(3):199–218, July 2000. 10.1023/A:1008191222954.

[45] Ruan Lakemond, Clinton Fookes, and Sridha Sridharan. Resection-Intersection Bundle Adjustment Revisited. *International Scholarly Research Network Machine Vision*, 2013:1–8, 01 2013. https://dx.doi.org/10.1155/2013/261956

[46] S. Lazebnik, Edmond Boyer, and J. Ponce. On Computing Exact Visual Hulls of Solids Bounded by Smooth Surfaces. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages I–156, 02 2001. https://dx.doi.org/10.1109/CVPR.2001.990469

[47] Kenneth Levenberg. A Method for the Solution of Certain Non-Linear Problems in Least Squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944.

[48] Eigen Library. http://eigen.tuxfamily.org, 2013.

[49] Peter Lindstrom. Triangulation Made Easy. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1554–1561, 2010. https://dx.doi.org/10.1109/CVPR.2010.5539785

[50] Manolis I. A. Lourakis and Antonis A. Argyros. SBA: A Software Package for Generic Sparse Bundle Adjustment. *ACM Transactions on Mathematical Software*, 36(1), mar 2009. https://dx.doi.org/10.1145/1486525.1486527

[51] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Compututer Vision*, 60(2):91–110, November 2004. https://dx.doi.org/10.1023/B:VISI.0000029664.99615.94

[52] Helmut Mayer. RPBA - Robust Parallel Bundle Adjustment Based on Covariance Information. *CoRR*, 2019. http://arxiv.org/abs/1910.08138

[53] Paul Merrell, Amir Akbarzadeh, Liang Wang, Philippos Mordohai, Jan-Michael Frahm, Ruigang Yang, David Nister, and Marc Pollefeys. Real-Time Visibility-Based Fusion of Depth Maps. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007. https://dx.doi.org/10.1109/ICCV.2007.4408984

[54] Paul C. Merrell, Philippos Mordohai, Jan-Michael Frahm, and Marc Pollefeys. Evaluation of Large Scale Scene Reconstruction. *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007.

[55] Duane Merrill. CUDA UnBound (CUB) Library, 2015. https://nvlabs.github.io/cub/.

[56] E.M. Mikhail, J.S. Bethel, and J.C. McGlone. *Introduction to Modern Photogrammetry*. Wiley, 2001.

[57] Yang Min. L-Infinity Norm Minimization in the Multiview Triangulation. In *Proceedings of the 2010 International Conference on Artificial Intelligence and Computational Intelligence: Part I*, AICI'10, page 488494. Springer-Verlag, Berlin, Heidelberg, 2010.

[58] P. Moreels and P. Perona. Evaluation of Features Detectors and Descriptors Based on 3D objects. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 1, pages 800–807 Vol. 1, 2005. https://dx.doi.org/10.1109/ICCV.2005.89

[59] Kai Ni, Drew Steedly, and Frank Dellaert. Out-of-Core Bundle Adjustment for Large-Scale 3D Reconstruction. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007. https://dx.doi.org/10.1109/ICCV.2007.4409085

[60] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, March/April 2008. https://dx.doi.org/10.1145/1365490.1365500

[61] D. Nistér. Reconstruction from Uncalibrated Sequences with a Hierarchy of Trifocal Tensors. In *Proc. of the European Conference on Computer Vision*, pages 649–663. Springer-Verlag, London, UK, 2000.

[62] D. Nistér. Frame Decimation for Structure and Motion. In *SMILE '00: Revised Papers from Second European Workshop on 3D Structure from Multiple Images of Large-Scale Environments*, pages 17–34. Springer-Verlag, London, UK, 2001.

[63] NVIDIA. CUDA C Programming Guide, version 11.7. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2022.

[64] Oxford Visual Geometry Group. Multi-View and Oxford Colleges Building Reconstruction. http://www.robots.ox.ac.uk/~vgg/, August 2009.

[65] Marc Pollefeys, Luc Van Gool, Maarten Vergauwen, Frank Verbiest, Kurt Cornelis, Jan Tops, and Reinhard Koch. Visual Modeling with a Hand-Held Camera. *International Journal of Computer Vision*, 59:207–232, 2004.

[66] Karthikeyan Natesan Ramamurthy, Chung-Ching Lin, Aleksandr Y. Aravkin, Sharath Pankanti, and Raphael Viguier. Distributed Bundle Adjustment. In *2017 IEEE International Conference on Computer Vision Workshops, ICCV Workshops 2017, Venice, Italy, October 22-29, 2017*, pages 2146–2154. IEEE Computer Society, 2017. https://dx.doi.org/10.1109/ICCVW.2017.251

[67] Shawn Recker, Mauricio Hess-Flores, and Kenneth I. Joy. Statistical Angular Error-Based Triangulation for Efficient and Accurate Multi-View Scene Reconstruction. In *Workshop on the Applications of Computer Vision (WACV)*, 2013.

[68] Maria I. Restrepo, Brandon A. Mayer, Ali O. Ulusoy, and Joseph L. Mundy. Characterization of 3-D Volumetric Probabilistic Scenes for Object Recognition. *IEEE Journal of Selected Topics in Signal Processing*, 6(5):522–537, 2012. https://dx.doi.org/10.1109/JSTSP.2012.2201693

[69] Susanna Ricco and Carlo Tomasi. Video Motion for Every Visible Point. In *2013 IEEE International Conference on Computer Vision*, pages 2464–2471, 2013. https://dx.doi.org/10.1109/ICCV.2013.306

[70] R. B. Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, May 9-13 2011.

[71] Jairo R. Sánchez, Hugo Álvarez, and Diego Borro. GFT: GPU fast triangulation of 3D points. In *Proceedings of the 2010 International Conference on Computer Vision and Graphics: Part II*, ICCVG'10, pages 235–242. Springer-Verlag, Berlin, Heidelberg, 2010.

[72] Jairo R. Sánchez, Hugo Álvarez, and Diego Borro. GPU Optimizer: A 3D Reconstruction on the GPU using Monte Carlo Simulations – How to Get Real Time without Sacrificing Precision. In *VISAPP 2009 - Proceedings of the Fifth International Conference on Computer Vision Theory and Applications, Angers, France, May 17-21, 2010 - Volume 1*, pages 443–446. INSTICC Press, 2010.

[73] Nadathur Satish, Mark Harris, and Michael Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proceedings of the 23rd IEEE International Parallel and*

Distributed Processing Symposium, May 2009. https://dx.doi.org/10.1109/IPDPS.2009.5161005

[74] D. Scharstein, R. Szeliski, and R. Zabih. A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. In *Proceedings of the IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV'01)*, SMBV '01, page 131. IEEE Computer Society, USA, 2001.

[75] Thomas Schps, Johannes L. Schnberger, Silvano Galliani, Torsten Sattler, Konrad Schindler, Marc Pollefeys, and Andreas Geiger. A Multi-view Stereo Benchmark with High-Resolution Images and Multi-camera Videos. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2538–2547, 2017. https://dx.doi.org/10.1109/CVPR.2017.272

[76] S.M. Seitz and C.R. Dyer. Photorealistic Scene Reconstruction by Voxel Coloring. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1067–1073, 1997. https://dx.doi.org/10.1109/CVPR.1997.609462

[77] Steven M. Seitz, Brian Curless, James Diebel, Daniel Scharstein, and Richard Szeliski. A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms. In *Proc. of the 2006 IEEE Conference on Computer Vision and Pattern Recognition*, pages 519–528, 2006.

[78] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo Tourism: Exploring Photo Collections in 3D. *ACM Trans. Graph.*, 25(3):835846, jul 2006. https://dx.doi.org/10.1145/1141911.1141964

[79] J. A. Snyman. *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*. Applied Optimization, Vol. 97, second edition. Springer-Verlag New York, Inc., 2005.

[80] Huaibo Song, Chenghai Yang, Jian Zhang, Clint Hoffmann, Dongjian He, and J. Thomasson. Comparison of Mosaicking Techniques for Airborne Images from Consumer-Grade

Cameras. *Journal of Applied Remote Sensing*, 10, 03 2016. https://dx.doi.org/10.1117/1.JRS.10.016030

[81] H. Stewenius, F. Schaffalitzky, and D. Nister. How Hard is 3-View Triangulation Really? In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 1, pages 686–693 Vol. 1, 2005. https://dx.doi.org/10.1109/ICCV.2005.115

[82] C. Strecha, W. von Hansen, L. Van Gool, P. Fua, and U. Thoennessen. On Benchmarking Camera Calibration and Multi-View Stereo for High Resolution Imagery. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008. https://dx.doi.org/10.1109/CVPR.2008.4587706

[83] R. Szeliski. Rapid Octree Construction from Image Sequences. *Computer Vision Graphics and Image Processing: Image Understanding*, 58(1):23–32, July 1993. http://dx.doi.org/10.1006/ciun.1993.1029

[84] Michael Van den Bergh, Xavier Boix, Gemma Roig, Benjamin Capitani, and Luc Van Gool. SEEDS: Superpixels Extracted via Energy-Driven Sampling. *International Journal of Computer Vision*, 111, 10 2012. https://dx.doi.org/10.1007/978-3-642-33786-4_2

[85] G. Vogiatzis, P.H.S. Torr, and R. Cipolla. Multi-View Stereo via Volumetric Graph-Cuts. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 391–398 vol. 2, 2005. https://dx.doi.org/10.1109/CVPR.2005.238

[86] Hoang-Hiep Vu, Patrick Labatut, Jean-Philippe Pons, and Renaud Keriven. High Accuracy and Visibility-Consistent Dense Multiview Stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(5):889–901, 2012. https://dx.doi.org/10.1109/TPAMI.2011.172

[87] Yichen Wei and Long Quan. Region-Based Progressive Stereo Matching. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recog-*

nition, 2004. CVPR 2004., volume 1, pages I–I, 2004. https://dx.doi.org/10.1109/CVPR.2004.1315020

[88] Andreas Wendel, Michael Maurer, Gottfried Graber, Thomas Pock, and Horst Bischof. Dense Reconstruction On-the-Fly. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1450–1457, 2012. https://dx.doi.org/10.1109/CVPR.2012.6247833

[89] Changchang Wu. SiftGPU: A GPU Implementation of David Lowe's Scale Invariant Feature Transform (SIFT). https://github.com/pitzer/SiftGPU, 2007.

[90] Changchang Wu. Towards Linear-Time Incremental Structure from Motion. In *Proceedings of the 2013 International Conference on 3D Vision*, 3DV '13, pages 127–134. IEEE Computer Society, Washington, DC, USA, 2013. https://dx.doi.org/10.1109/3DV.2013.25

[91] Changchang Wu, S. Agarwal, B. Curless, and S. M. Seitz. Multicore Bundle Adjustment. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '11, pages 3057–3064. IEEE Computer Society, USA, 2011. https://dx.doi.org/10.1109/CVPR.2011.5995552

[92] Tai-Pang Wu, Sai-Kit Yeung, Jiaya Jia, and Chi-Keung Tang. Quasi-Dense 3D Reconstruction Using Tensor-Based Multiview Stereo. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1482–1489, 2010. https://dx.doi.org/10.1109/CVPR.2010.5539796

[93] Ruigang Yang and Marc Pollefeys. Multi-Resolution Real-Time Stereo on Commodity Graphics Hardware. In *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR'03, pages 211–217. IEEE Computer Society, Washington, DC, USA, 2003.

[94] C. Zach. Fast and High Quality Fusion of Depth Maps. In *International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT)*, volume 1, 2008.

[95] Runze Zhang, Siyu Zhu, Tian Fang, and Long Quan. Distributed Very Large Scale Bundle Adjustment by Global Camera Consensus. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 29–38, 2017. https://dx.doi.org/10.1109/ICCV.2017.13

[96] Z. Zhang and Y. Shan. A Progressive Scheme for Stereo Matching. In *Revised Papers from Second European Workshop on 3D Structure from Multiple Images of Large-Scale Environments*, SMILE '00, pages 68–85. London, UK, 2001.

[97] Maoteng Zheng, Shunping Zhou, Xiaodong Xiong, and Junfeng Zhu. A New GPU Bundle Adjustment Method for Large-Scale Data. *Photogrammetric Engineering & Remote Sensing*, 83(9), 2017.

[98] K. Zhu, M. Butenuth, and P. d'Angelo. Computational Optimized 3D Reconstruction System for Airborne Image Sequences. In *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, ISPRS Comm. III*, volume XXXVIII, 2010.