A Comparative Study of GPU-Accelerated Multi-View Sequential Reconstruction Triangulation Methods for Large-Scale Scenes

Jason Mak, Mauricio Hess-Flores, Sean Recker, John D. Owens, Kenneth I. Joy

University of California, Davis

Abstract. The angular error-based triangulation method and the parallax path method are both high-performance methods for large-scale multi-view sequential reconstruction that can be parallelized on the GPU. We map parallax paths to the GPU and test its performance and accuracy as a triangulation method for the first time. To this end, we compare it with the angular method on the GPU for both performance and accuracy. Furthermore, we improve the recovery of path scales and perform more extensive analysis and testing compared with the original parallax paths method. Although parallax paths requires sequential and piecewise-planar camera positions, in such scenarios, we can achieve a speedup of up to 14x over angular triangulation, while maintaining comparable accuracy.

1 Introduction

Recently, there has been a great deal of work dealing with multi-view reconstruction of scenes, for example in applications such as robotics, surveillance and virtual reality. One specific scenario for reconstruction is 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.

For aerial reconstruction, and reconstruction in general, performance scalability is a growing concern. Improved technology has allowed for the collection of numerous images at very high resolutions. To address this problem, researchers have developed new algorithms that can perform faster while yielding accurate reconstructions. Another approach is to leverage modern hardware, specifically parallel architectures, that include Graphics Processing Units (GPUs), which are now widely used to speed up a variety of computational problems. GPU hardware is designed to compute large amounts of work simultaneously, which makes GPUs ideal for high-performance image processing.

To this end, the parallax paths method is a promising framework developed by Hess-Flores et al. [1] 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.

The main advantage of this method 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. [1].

The main contribution of this paper is to 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 [2], which we refer to from now on as *fast triangulation*. Recker's triangulation method 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. Parallax paths is more parallelizable and performs faster than fast triangulation, and with good starting feature tracks and the camera path as a reliable constraint, we can obtain comparable accuracy. Furthermore, the effect of different path scales is further defined and evaluated with respect to the original method. In the first section, we discuss related work, then in the following sections, we provide an overview of the fast triangulation method and the parallax paths method, including new insights for the latter. Next, we discuss GPU implementations, show the results of our comparison experiments, and finally end with our conclusions.

2 Related Work

The input to scene reconstruction is typically a set of images and in some cases camera calibration information, while the output is typically a 3D point cloud along with color and/or normal information, representing scene structure. For general reconstruction algorithms in the literature, there are comprehensive overviews and comparisons given in Seitz et al. [3] and Strecha et al. [4]. As for

sequential reconstruction algorithms, Pollefeys et al. [5] provides a method for reconstruction from hand-held cameras, Nistér [6] deals with reconstruction from trifocal tensor hierarchies, while Fitzgibbon et al. [7] provides an approach for turntable sequences. State-of-the-art software packages such as *VisualSfM* [8] and *Bundler* [9] provide very accurate feature tracking, camera poses and scene structure, based mainly on sparse feature detection and matching, such as with the SIFT algorithm [10] and others inspired by its concept.

This paper focuses on one of the final stages of reconstruction, known as *triangulation* [11], where 3D positions for scene points are computed. The accuracy of triangulation is a function of previously-computed feature tracking, camera intrinsic calibration, and pose estimation [11]. Typically, 3×4 projection matrices are used to encapsulate all camera intrinsic and pose information. The most widely-used method in the literature is *linear triangulation* [11], where a system of the form $AX = 0$ is solved by eigen-analysis or Singular Value Decomposition (SVD). The data matrix A is a function of feature track and camera projection matrix values. The obtained solution is a direct, best-fit, and non-optimal solve, where numerical stability issues arise with near-parallel cameras. Another simple method is the midpoint method [11], but it is very inaccurate in general. A second class of algorithms is based on optimizing a cost function based on an initial direct solution. In general, these methods lack solid experimental results as far as error and processing time against different noise and camera configurations. Agarwal et al. [12] use fractional programming and a *branch and bound* algorithm to determine the global optimum. Hartley and Kahl [13] as well as Min [14] perform convex optimization on an L_∞ cost function making use of second-order cone programming (SOCP). Dai et al. [15] use a L_∞ optimization method based on gradually contracting a region of convexity towards computing the optimum.

Additional triangulators have been developed with not only accuracy but also performance scalability in mind. Recker et al.'s *fast triangulation* method [2] obtains an initial position through the midpoint method and applies adaptive gradient descent [16] on an angular error-based L_1 cost function. This function is shown to have a large basin in the vicinity of the global optimum, which avoids converging to unwanted local minima. Also, the L_1 cost function is more robust to outliers than the L_2 norm of reprojection error. Furthermore, it introduces a statistical sampling component to increase efficiency without sacrificing accuracy. This results in a significant speed increase and better reprojection errors than with other triangulators, such as linear triangulation. Hess-Flores [1] developed the parallax paths method for reconstruction in scenarios where the camera path can be modeled by piecewise-planar segments. The procedure results in an updated set of feature tracks, such that the speed and accuracy of reconstruction and bundle adjustment is improved. However, it was not tested as a standalone triangulator nor compared against any other known triangulation algorithms for speed or accuracy. Furthermore, no attempts at parallelization were made. Sánchez et al. [17, 18] developed a triangulator on the GPU using an algorithm based on Monte Carlo simulations. They achieve good speedup over Levenberg-Marquardt [19] and have comparable accuracy. However, their implementation was

not tested on large-scale data. Mak et al. [20] developed a GPU implementation of Recker's fast triangulation method and achieve up to 40x speedup over a CPU implementation on large-scale data. To the best of our knowledge, this is one of the fastest GPU triangulators in existence that still maintains good accuracy. In this work, we implement parallax paths on the GPU for the first time, and compare it with the GPU fast triangulation method for speed and accuracy.

3 A Summary of Fast Triangulation

Recker et al. [2] propose a L_1 triangulation cost function based on an angular error measure for a candidate 3D position, p, with respect to its feature track t. Its inputs are feature tracks across N images and their respective 3×4 camera projection matrices P_i. For each camera in a track, the cost function measures the dot product between the ray from the camera center C_i to p, known as unit vector v_i, and the ray from C_i through its 2D feature t, known as w_{ti}. This concept is displayed in Fig. 1(a). The total cost function value is the sum of these dot products across all cameras, subtracted from the total number of cameras to ensure that the absolute global minimum is zero, and then averaged. Dot products can vary from $[-1, 1]$, but only points that lie in front of the cameras need to be taken into account, corresponding to the range $[0, 1]$. Given C_i cameras, the set of all feature tracks T, and a 3D evaluation position $p = (X, Y, Z)$, the cost function for p with respect to a track $t \in T$ is given by Eq. 1 [2].

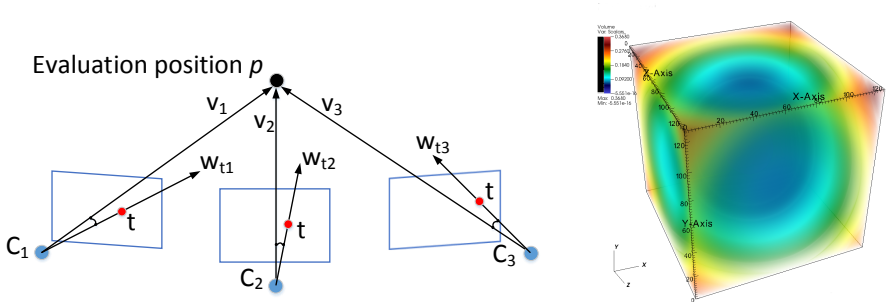
$$f_{t \in T}(p) = \frac{\sum_{i \in I}(1 - \hat{v_i} \cdot \hat{w_{ti}})}{||I||} \quad . \tag{1}$$

In their nomenclature, $I = \{C_i | t$ "appears in" $C_i\}$, $\boldsymbol{v_i} = (p - C_i)$, and $\boldsymbol{w_{ti}} = P_i^+ t_i$. The right pseudo-inverse of P_i is given by P_i^+, while t_i is the homogeneous coordinate of track t in camera i. The normalized vectors are defined as $\hat{v_i} = \frac{v_i}{||v_i||}$ and $\hat{w_{ti}} = \frac{w_{ti}}{||w_{ti}||}$. Gradient values are defined in Eqs. 5–7 of Recker et al. [2].

To visualize the smooth variation of this L_1 cost function, as analyzed and discussed in Recker et al. [2], Fig. 1(b) shows a scalar field, where each dense grid position encodes the cost function value at that specific 3D location. Values are color-coded such that redder values represent higher cost function values, while blue represents values closer to zero. Notice that in this particular case, the global minimum, not explicitly displayed, lies in the center of the displayed bounding box. The variation within this box, however, is very smooth, indicating that the function is a sink with convergence likely even from distant locations. This allows for simple methods such as adaptive gradient descent [16] to be used for optimization, starting from an initial midpoint estimate [2]. For more details on the overall method, the reader is referred to Recker et al. [2].

3.1 Degeneracies in Fast Triangulation

There are specific degeneracies that can affect fast triangulation. The first is an initial midpoint estimate which is very inaccurate. Despite the sink behavior of



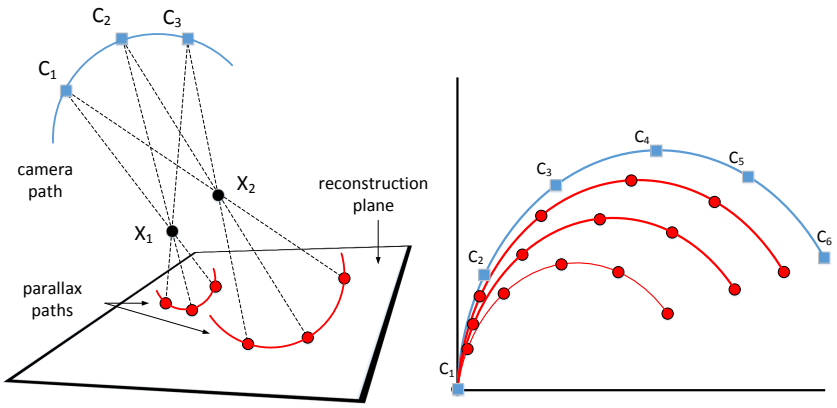
(a) Fast triangulation method (b) Scalar field

Fig. 1. (a) In fast triangulation, rays are shot through a candidate point and through feature locations. (b) A volume view of a scalar field representing an L_1 cost function [2] evaluated at a dense grid inside a bounding box encasing a position in the reconstruction, with blue closer to zero cost.

the cost function, a very inaccurate starting position can lead adaptive gradient descent in the wrong direction. Though Recker et al. [2] 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 [22] can be used for this purpose.

4 Parallax Paths—A Further Analysis

The parallax paths method was developed by Hess-Flores et al. [1] to help yield more accurate 3D reconstructions in aerial and turntable sequences. However, the concept of scale was not generalized or further analyzed, and this paper helps enhance and improve on this concept. 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 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. 2(a). The concept of scale is easily visualized when translating all parallax paths to a 2D position-invariant reference, as shown in Fig. 2(b).



(a) Parallax paths method (b) 2D position-invariant reference

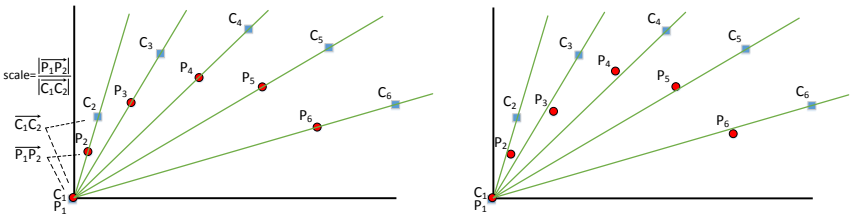
Fig. 2. (a) Rays from cameras $C_i \dots C_n$ through a scene point X_i intersect a plane, creating a parallax path, which is a scaled version of the camera path. Points closer to the cameras create bigger paths. (b) The camera path and parallax paths are translated to a position-invariant reference, with a track's path origin coinciding with the anchor camera for the track.

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. 2(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. 3(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. The original work by Hess-Flores et al. [1] did not mathematically define a direct way to obtain this scale, and we provide an efficient way to compute its value. For the first locus line in Fig. 3(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. 2.

$$scale = \frac{|\overrightarrow{P_1P_2}|}{|\overrightarrow{C_1C_2}|} = \frac{|\overrightarrow{P_1P_3}|}{|\overrightarrow{C_1C_3}|} = \frac{|\overrightarrow{P_1P_4}|}{|\overrightarrow{C_1C_4}|} = \frac{|\overrightarrow{P_1P_5}|}{|\overrightarrow{C_1C_5}|} = \dots = \frac{|\overrightarrow{P_1P_N}|}{|\overrightarrow{C_1C_N}|} \quad . \tag{2}$$

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.



(a) Perfect tracks (b) General case

Fig. 3. (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.

4.1 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 Fig. 3(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 Fig. 2(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. 2 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 [1]. 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 does not 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, so using adjacent cameras is less of a problem.

The parallax paths method is very powerful because it provides additional constraints to yield an accurate reconstruction, which are not present in bundle adjustment [19] 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* [8] and *Bundler* [9] 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. 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.

5 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. As an overview of the model, CUDA *threads* are divided into *blocks* that run in parallel, each of which contains up to 1024 threads. Each thread runs the same CUDA program called a *kernel*. Blocks of threads are assigned to *streaming multiprocessors* (SMs). An SM runs a group of 32 threads called *warps* in a SIMD manner. Like a CPU, a GPU has a large, slow main memory, as well as caches. A faster, more local memory called shared memory is also available that allows threads in the same block to share data. Certain types of applications are more suitable for the GPU. They must be parallel enough to

require an enormous number of threads, and since threads within a warp run in SIMD, the program control flow must not cause threads to frequently diverge (perform different operations due to conditional statements). A more detailed description of the CUDA programming model is given by Nickolls et al. [23].

5.1 Fast Triangulation GPU Implementation

Mak et al. [20] 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. The authors 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 [20], which must be a size that is a multiple of the warp size (32).

5.2 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 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. Fig. 4 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.

Fig. 4. Parallax paths stages on the GPU, including parallelism P per stage.

6 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.

6.1 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.

Fig. 5(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, Fig. 5(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.

Fig. 5(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. *PP2* is left out here because it only uses 2 cameras regardless of track length. Fig. 5(b) shows that *PP* also scales better than *FT* with increasing cameras.

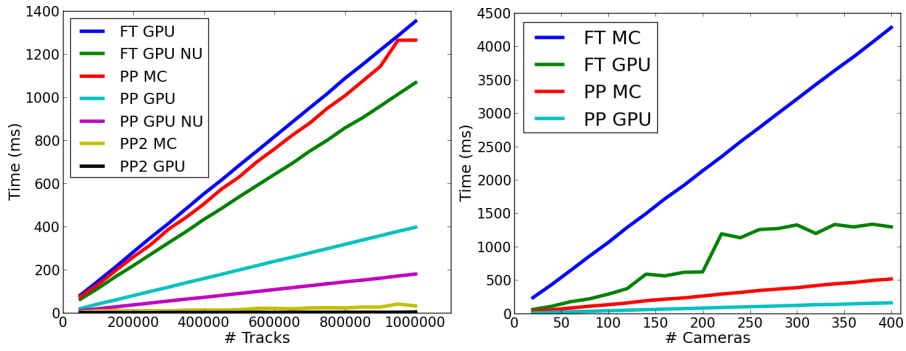


Fig. 5. (a) Runtime performance with an increasing number of tracks. The number is increased up to 1,000,000, in increments of 50,000. Track length is fixed at 100 cameras, except for the *NU* cases, where it is varied from 2–100. (b) Runtime performance with an increasing number of cameras. Cameras are varied up to 400, in increments of 50, and track length is fixed at 100,000.

In the error tests shown in Fig. 6, 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.

Fig. 6(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*. Fig. 6(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.

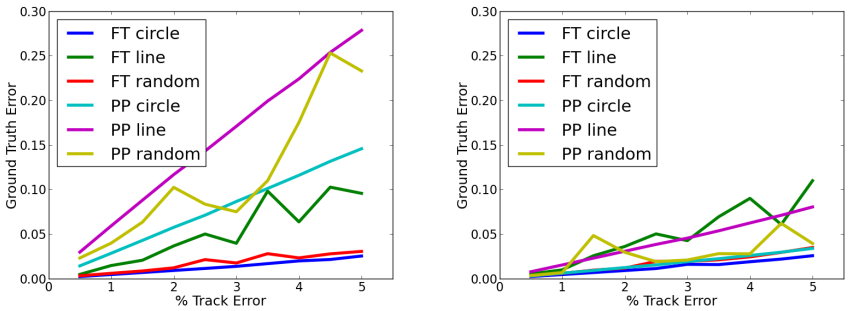


Fig. 6. Ground truth error vs. feature track error for synthetic data. (a) All features in each track subject to error. (b) No error in first feature of each track.

6.2 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 [1]. 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 1(a) displays results for fast triangulation (FT), Table 1(b) for parallax paths (PP), and Table 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.

Table 1. Times in milliseconds for serial, multi-core, 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. Some runtimes for *Horse* are left out because they were too small to measure.

(a) Fast triangulation

Data set	N	C	serial	multicore	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	multicore	GPU	Speedup	ϵ	Speedup 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	–

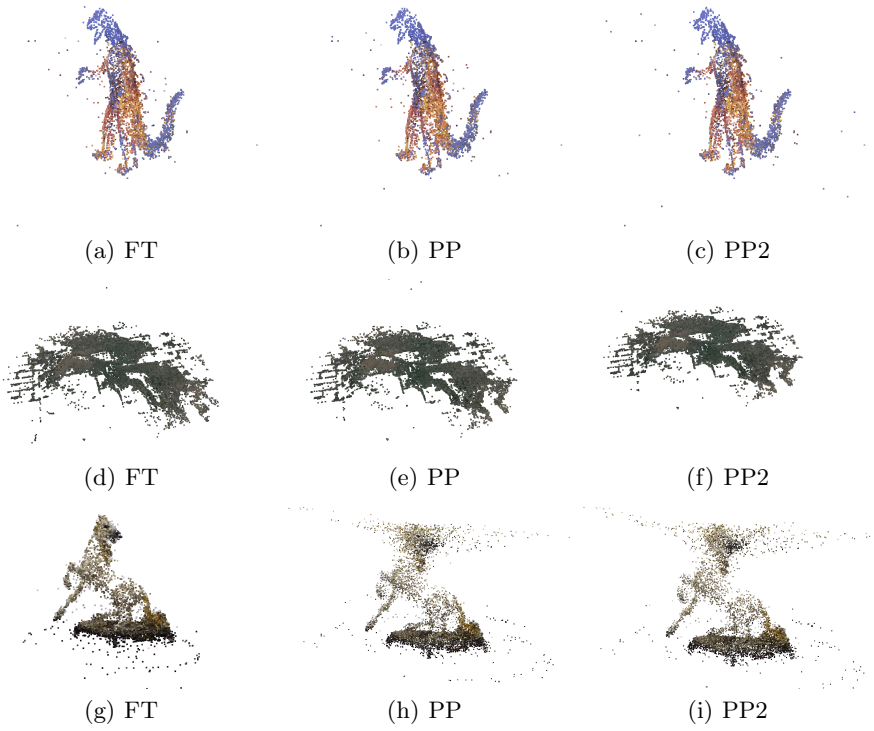
(c) Parallax paths first 2 cameras

Data set	N	C	serial	multicore	GPU	Speedup	ϵ	Speedup vs. FT
Dinosaur [24]	4983	36	2	0.5	0.07	28x	1.246	42x
Canyon [2]	103,153	90	37	7	0.36	102x	0.863	39x
Canyon Dense [2]	997,115	2	351	64	3	117x	1.847	7x
Horse [25]	9509	73	3.4	0.8	–	–	1.232	–

Finally, Fig. 7(a)-(c) shows the reconstruction of the *Dinosaur* dataset [24] using respectively *FT*, *PP*, and *PP2* from left to right. Fig. 7(d)-(f) displays the same but for the *Canyon* dataset [2]. 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, Fig. 7(g) shows a good reconstruction obtained from *FT* for the *Horse* [25] dataset, whereas Fig. 7(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.

7 Conclusion

In this paper, 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



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

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

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

Fig. 7. Reconstructions of three scenes: (a)-(c) *Dinosaur* [24]. (d)-(f) *Canyon* [2]. (g)-(i) *Horse* [25]. 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*.

contributions of the paper 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 develop the parallax paths further than in 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 accurate 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.

References

1. Hess-Flores, M., Duchaineau, M.A., Joy, K.I.: Sequential reconstruction segment-wise feature track and structure updating based on parallax paths. In: Proceedings of the 11th Asian Conference on Computer Vision - Volume Part III. ACCV'12, Berlin, Heidelberg, Springer-Verlag (2013) 636–649
2. Recker, S., Hess-Flores, M., Joy, K.I.: Statistical angular error-based triangulation for efficient and accurate multi-view scene reconstruction. In: Workshop on the Applications of Computer Vision (WACV). (2013) 68–75
3. Seitz, S.M., Curless, B., Diebel, J., Scharstein, D., Szeliski, R.: A comparison and evaluation of multi-view stereo reconstruction algorithms. In: CVPR '06, Washington, DC, USA, IEEE Computer Society (2006) 519–528
4. Strecha, C., von Hansen, W., Gool, L.J.V., Fua, P., Thoennessen, U.: On benchmarking camera calibration and multi-view stereo for high resolution imagery. In: Proc. of the 2008 IEEE Conference on Computer Vision and Pattern Recognition. (2008)
5. Pollefeys, M., Van Gool, L., Vergauwen, M., Verbiest, F., Cornelis, K., Tops, J., Koch, R.: Visual modeling with a hand-held camera. International Journal of Computer Vision **59** (2004) 207–232
6. Nistér, D.: Reconstruction from uncalibrated sequences with a hierarchy of trifocal tensors. In: ECCV '00, London, UK, Springer-Verlag (2000) 649–663
7. Fitzgibbon, A.W., Cross, G., Zisserman, A.: Automatic 3D model construction for turn-table sequences. In: Proceedings of the European Workshop on 3D Structure from Multiple Images of Large-Scale Environments, London, UK, Springer-Verlag (1998) 155–170
8. Changchang Wu: VisualSfM: A visual structure from motion system (2011)
9. Snavely, N., Seitz, S.M., Szeliski, R.: Photo tourism: exploring photo collections in 3d. In: ACM SIGGRAPH '06, New York, NY, USA, ACM (2006) 835–846
10. Lowe, D.: Distinctive image features from scale-invariant keypoints. International Journal On Computer Vision **60** (2004) 91–110
11. Hartley, R.I., Zisserman, A.: Multiple View Geometry in Computer Vision. 2nd edn. Cambridge University Press (2004)
12. Agarwal, S., Chandraker, M.K., Kahl, F., Kriegman, D., Belongie, S.: Practical global optimization for multiview geometry. In: Proc. of the 9th European Conference on Computer Vision. (2006) 592–605
13. Hartley, R., Kahl, F.: Optimal algorithms in multiview geometry. In: Proc. of the 8th Asian conference on Computer Vision – Volume Part I. ACCV '07 (2007) 13–34
14. Min, Y.: L-infinity norm minimization in the multiview triangulation. In: Proc. of the 2010 International Conference on Artificial Intelligence and Computational Intelligence: Part I. AICI '10 (2010) 488–494
15. Dai, Z., Wu, Y., Zhang, F., Wang, H.: A novel fast method for L_∞ problems in multiview geometry. In: Proc. of the 12th European Conference on Computer Vision – Volume Part V. ECCV '12 (2012) 116–129
16. Snyman, J.A.: Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms. Second edn. Applied Optimization, Vol. 97. Springer-Verlag New York, Inc. (2005)
17. Sánchez, J.R., Álvarez, H., Borro, D.: GFT: GPU fast triangulation of 3D points. In: Proc. of the 2010 International Conference on Computer Vision and Graphics: Part II. ICCVG '10, Berlin, Heidelberg, Springer-Verlag (2010) 235–242

18. Sánchez, J.R., Álvarez, H., Borro, D.: GPU optimizer: A 3D reconstruction on the GPU using Monte Carlo simulations – how to get real time without sacrificing precision. In: Proc. of the 2010 International Conference on Computer Vision Theory and Applications. (2010) 443–446
19. Lourakis, M.I.A., Argyros, A.A.: The design and implementation of a generic sparse bundle adjustment software package based on the Levenberg-Marquardt algorithm. Technical Report FORTH-ICS TR-340-2004, Institute of Computer Science – FORTH (2004)
20. Mak, J., Hess-Flores, M., Recker, S., Owens, J.D., Joy, K.I.: GPU-accelerated and efficient multi-view triangulation for scene reconstruction. In: Workshop on the Applications of Computer Vision (WACV). (2014)
21. Fischler, M.A., Bolles, R.C.: Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Readings in Computer Vision: Issues, Problems, Principles, and Paradigms (1987) 726–740
22. Nistér, D.: 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, London, UK, Springer-Verlag (2001) 17–34
23. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. ACM Queue (2008) 40–53
24. Oxford Visual Geometry Group: Multi-view and Oxford Colleges building reconstruction (2009) `http://www.robots.ox.ac.uk/~vgg/`.
25. Moreels, P., Perona, P.: Evaluation of features detectors and descriptors based on 3d objects. In: Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1 - Volume 01. ICCV '05, Washington, DC, USA, IEEE Computer Society (2005) 800–807