Title
GPU-Accelerated and Efficient Multi-View Triangulation for Scene Reconstruction

Authors
Mak, Jason
Hess-Flores, Mauricio
Recker, Sean
et al.

Publication Date
2014

Peer reviewed

GPU-Accelerated and Efficient Multi-View Triangulation for Scene Reconstruction

Jason Mak[1], Mauricio Hess-Flores[2], Shawn Recker[3], John D. Owens[4], Kenneth I. Joy[5]
University of California, Davis
([1]jwmak, [2]mhessf, [3]strecker, [4]jowens)@ucdavis.edu, [5]kenneth.i.joy@gmail.com

Abstract

This paper 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 40x 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.

1. Introduction

Triangulation is a key step in multi-view scene reconstruction. It 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 [11] 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 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. To this end, linear triangulation [11] is a fast method to solve for 3D points based on linear least squares, but given noisy inputs, the final result can be very inaccurate. The midpoint method [11] 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.

A number of optimal algorithms have since been developed, some of which involve direct, closed-form solutions and others which seek to optimize a cost function, as described in Section 2. Current direct *polymonial methods* solve for just two or three views [5, 10, 12, 13, 25]. A polynomial solver has not been achieved for more than three views. Multi-view triangulation has traditionally been treated in two phases, where an initial linear method such as N-view linear triangulation [11] 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 [1]. This procedure is prone to local minima, so a very accurate initialization is required. A triangulator presented by Recker *et al*. [18] optimizes a novel L_1-based error cost function derived from the angles between camera rays and is smoothly-varying in a large vicinity near the global optimum. The cost function is optimized with adaptive gradient descent [24] given an initial estimate. It shows a significant performance increase and better final reprojection errors than other triangulators, such as N-view linear triangulation, but is not provably optimal. A few N-view optimal triangulators have been proposed, but with very limited results. The work of Agarwal *et al*. [3] is based on fractional programming and branch-and-bound theory. There are also a few methods based on convex optimization on an L_∞ cost function, such as Hartley and Kahl [9], Min [14], and most recently Dai *et al*. [7]. However, 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*. [7]. 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. Graphics Processing Units (GPUs) are increasingly ubiquitous in high-performance parallel computing. Originally designed to accelerate graphics, GPUs are now widely used as acceleration tools for a variety of applications. NVIDIA's CUDA Programming Model [16] has greatly expedited this transition, allowing developers to write general purpose programs for the GPU in a language based on C/C++. GPUs are well-suited for image processing and computer vision problems due to their high memory bandwidth, which enables efficient access of large image data, and their ability to exploit data parallelism. Additionally, numerous cores allow for a divide and conquer approach on high-resolution images. Furthermore, GPUs are becoming more widely found in smartphones and tablets, bringing their computational power into the hands of ordinary users. Programming GPUs predominantly follows the Single-Instruction-Multiple-Data (SIMD) model, where multiple threads run the same operations on different data in lockstep. An algorithm to be accelerated by the GPU must map well to this computational model. Sánchez *et al*. [20,21] 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.

The main contribution of this paper is to introduce a fast and accurate GPU N-view triangulator. It is based on the cost function of Recker *et al*. [18], which is ideal for parallelization because it consists of abundant independent 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. Furthermore, 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 faster than contemporary serial and multi-core CPU implementations, 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.

2. Related work

Multi-view scene reconstruction involves a number of stages applied sequentially, where the output of one stage directly affects accuracy in the following steps. A comprehensive overview and comparison of different methods is given in Strecha *et al*. [26]. Triangulation is one of the final stages in reconstruction and its accuracy is a direct 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]. 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, and non-optimal solve, regardless of how noisy the inputs are. Numerical stability issues are possible, especially with near-parallel cameras. A recent triangulator by Recker *et al*. [18] proved to be faster and more efficient than linear triangulation. The method obtains an initial position through the midpoint method and applies adaptive gradient descent [24] 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, 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. However, the results are not provably optimal, and rely on a possibly inaccurate midpoint-based initialization.

There exist a number of optimal triangulation algorithms in the literature. One class of algorithms is based on *polynomial methods* [9], 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 [25]. This im-

plies 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 [10] 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 [13] 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. Stewénius *et al*. [25] 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*. [5] 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. 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*. [3] use fractional programming and a *branch and bound* algorithm to find a position arbitrarily close to the global optimum. Hartley and Kahl [9] as well as Min [14] perform convex optimization on an L_∞ cost function making use of second-order cone programming (SOCP). Dai *et al*. [7] describe an L_∞ optimization method based on gradually contracting a region of convexity towards computing the optimum. Sánchez *et al*. [20, 21] present an algorithm based on Monte Carlo simulations, performed on a GPU. Compared against Levenberg-Marquardt [1], it achieves the same precision but in much less time. In the next section, we propose an N-view triangulator, which outperforms the existing algorithms in speed while yielding comparable reprojection errors, though it is not provably optimal. It can be applied on an arbitrary number of cameras and does not suffer from numerical stability or precision issues.

3. GPU-accelerated triangulation

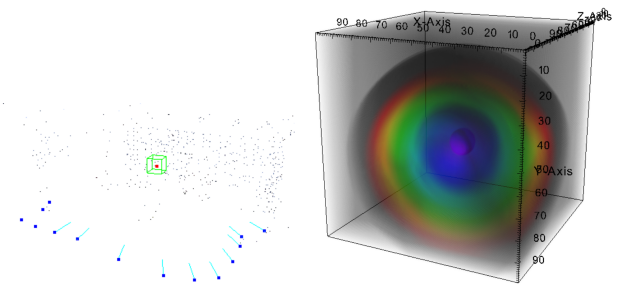
3.1. Triangulation cost function

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*. [18] proposed an 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 a set of feature tracks across N images and their respective 3×4 camera projection matrices P_i. The error for p is computed 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 $[0, 1]$. We use the same nomenclature as in Recker *et al*. [18] to define the cost function. Given C_i cameras, T the set of all feature tracks, and $p = (X, Y, Z)$ a 3D evaluation position, the cost function for p with respect to a track $t \in T$ is displayed in Eq. 1.

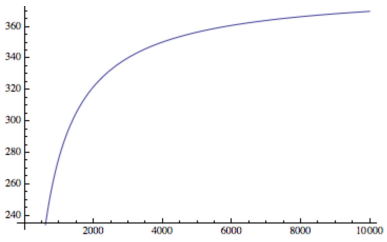
$$f_{t \in T}(p) = \frac{\sum_{i \in I}(1 - \hat{v_i} \cdot \hat{w_{ti}})}{||I||} \tag{1}$$

Here, $I = \{C_i | t \text{ "appears in" } C_i\}$, $\vec{v_i} = (p - C_i)$, 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 $\hat{w_{ti}} = \frac{w_{ti}}{||w_{ti}||}$. Gradient values are defined in Eqs. 5–7 of Recker *et al*. [18].

To analyze the convexity properties of this function, we step away from a purely mathematical approach and apply a more practical procedure. Fig. 1(b) shows a scalar field, consisting of the L_1 cost function measured for a dense set of test positions near a known ground-truth position in Fig. 1(a). 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 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. Intuitively, a dot product can vary 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, such as the the cost function of Eq. (1), shown rendered via a scalar field in Fig. 1(b). Optimization is performed with adaptive gradient descent [24], starting from an initial midpoint estimate [18].



(a) Bounding box (b) Scalar field



(c) Sample size

Figure 1: (a) Multi-view reconstruction of the *castle-P19* dataset [26], with cameras in dark blue. (b) A volume view of a scalar field representing an L_1 cost function [18] evaluated at a dense grid inside a bounding box encasing a position in the reconstruction, with purple closer to zero cost. (c) Sample size (y-axis) using Cochran's formula [6] with a 95% confidence level on different population sizes (x-axis).

3.1.1 Statistical sampling

We use a statistical sampling procedure to choose a statistically meaningful sample of rays as opposed to the entire available set, N. This procedure differs from RANSAC [8] since we do not seek to fit a model, and therefore deal with outliers based on statistics. We use Cochran's formula [6] to compute sample size n_0, as shown in Eq. 2. 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. The value for 'd' 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 [6] should be used to calculate the final sample size, n, as shown in Eq. 2. In our algorithm a 95% confidence level with a 5% margin of error is used. Notice in Fig. 1(c) that sample size stabilizes with large numbers, which is key towards our algorithm's speed.

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

3.1.2 Initialization based on clustering

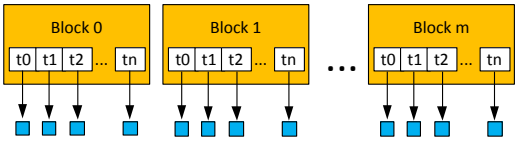
It is worth noting that we attempted implementing a much more robust and exhaustive initialization than that of Recker *et al.* [18], which is a simple midpoint start with a fixed threshold. First, the total possible number of pairs among a sample of N cameras is computed as $N(N-1)/2$. Next, the midpoint algorithm is used to compute a point between every camera pair from the sample. Clustering is then 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 an important result that the original initialization method is in general better because of speed and equal accuracy than this seemingly more robust procedure.

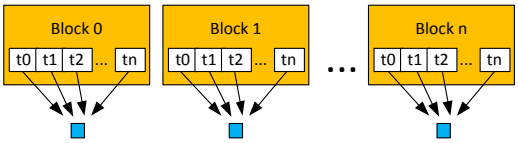
3.2. GPU implementation

Nickolls *et al.* [15] describes the CUDA programming model in detail; we now highlight the features of CUDA that are most relevant for our work. 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 many *streaming multiprocessors* (SMs), which runs groups of 32 threads called *warps* in lockstep 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*). 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.

Our contribution is a triangulation algorithm which 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 3.2.1 and 3.2.2. The simple approach is to parallelize across tracks and triangulate each track independently in a separate 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.



(a) One thread per track



(b) One block per track

Figure 2: Two approaches to parallelizing our triangulator.

3.2.1 One Thread Per Track

The first implementation, shown in Fig. 2(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 [22]. We use the radix sort routine from the GPU Thrust library [4] for sorting. Sorting can reduce the divergence within warps, thereby improving performance. Fig. 3(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 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 Fig. 2(b), is more suitable for data with long feature tracks. 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 modern GPUs, each thread block has access to 48 KB 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.

4. 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 K40, which features 15 SMs, for a total of 2880 cores. Tesla GPUs have improved performance for double-precision arithmetic, a feature we use in our triangulator. We find that the different double-precision support in GPU and CPU architectures leads to no significant differences in results for the triangulation algorithm under consideration. To parallelize the triangulator on the multi-core 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 [2]. 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.

4.1. Synthetic tests

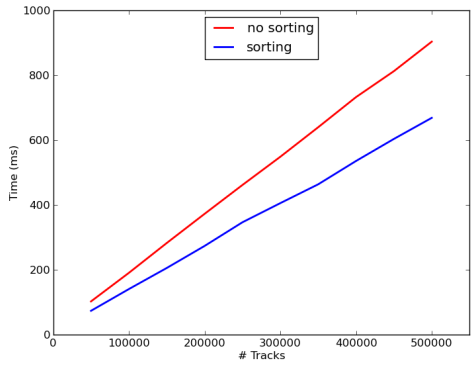
The first test on synthetic data measures the processing times as the number of tracks is increased, for the GPU implementation that assigns one thread per track vs. the multicore CPU implementation. Track count is increased in increments of 50,000, while a fixed length of 100 is used for all tracks. 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 Fig. 3(b). The performance of the GPU scales better than that of the multicore 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 in the Internet, where images are not acquired sequentially. Track length is fixed at 100, and the number of tracks is fixed at 10,000. Fig. 4(a) shows that runtime is hardly affected with small increases of track error.

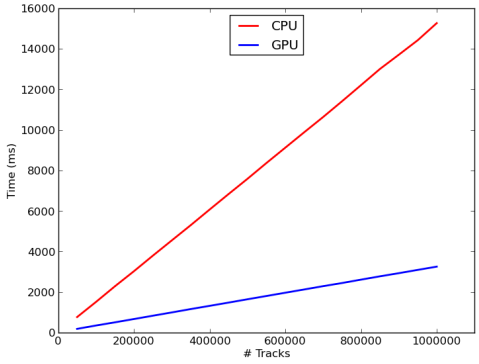
Finally, we compare the performance of the two GPU implementations. A variable track length from 10–100 is used. The number of tracks tested were 20,000 and 100,000. In Fig. 4(b), the performance crossover point occurs at a track length of 21. Since a thread block always consists of a multiple of 32 threads (a warp), when the sample size is not a multiple of 32, extra threads are idle in one-block-per-track. As track length increases, the performance penalty for idling threads becomes smaller because there are more warps per block, and therefore a lower fraction of threads idling within a block. In addition, the use of shared memory to store track and camera data in one-block-per-track is more beneficial with longer track lengths. However, when track lengths are short, one-thread-per-track performs better even with a large number of tracks. One-thread-per-track works better for track lengths less than 21 and with lots of tracks. Otherwise, one-block-per-track is more scalable.

4.2. Evaluation on real data

For real scenes, processing time and reprojection error were evaluated, as displayed in Table 1. Datasets featuring different scene types were evaluated. For the GPU implementation, we use the one-thread-per-track implementation with sorting because most of the tracks in the data



(a)



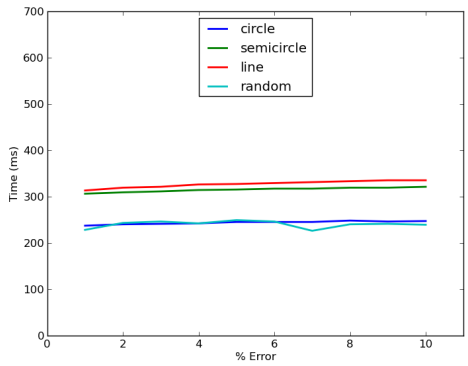
(b)

Figure 3: (a) GPU runtime performance with and without sorting for datasets with an increasing number of tracks. Track lengths in a dataset vary from 1 to 100. (b) Performance of a multicore CPU vs. a GPU for an increasing number of tracks. Track lengths are fixed at 100.

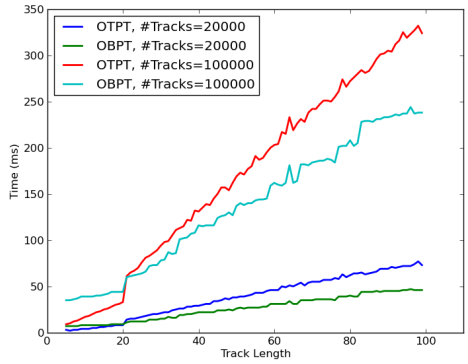
did not exceed 100 cameras and usually had varying length, except for *Brown12*, where one-block-per-track was used. We found that the GPU implementation was at best approximately 9 times faster than multi-core and 40 times faster than serial. The general trend shows greater speedups as track count increases, more importantly so than the variation in track length. In the specific cases of *Dinosaur* and *Brown12*, the multi-core CPU implementation is faster than the GPU, due to the track lengths and the number of points being too small to truly utilize the throughput capabilities of the GPU. No track length is greater than 21 in *Dinosaur*. In contrast, in *Canyon dense* all tracks are of length 2, but there are hundreds of thousands of them, leading to a large speedup. When comparing to other triangulators, Stewénius *et al*. [25] took 20 hours on the *Dinosaur* dataset [17] and Byröd *et al*. took 2.5 minutes, but ours takes less than 4 ms. Finally, obtained reconstructions are shown in Fig. 5.

Data set	N	C	t_{serial}	t_{mc}	t_{gpu}	S_{mc}	S_{gpu}	ϵ	ϵ_L
Brown12 [19]	4429	337	51	15	17	3.4x	3.0x	1.541	1.405
Dinosaur [17]	4983	36	9	2	4	4.5x	2.3x	0.467	0.477
Canyon [18]	103,153	90	288	86	16	3.3x	18x	0.226	0.231
Canyon Dense [18]	997,115	2	1440	342	36	4.2x	40x	1.838	1.841
Palmdale Distorted	58,500	66	244	59	7	4.1x	35x	1.138	1.713
City	16,179	10	110	31	4	3.5x	27x	0.726	0.982

Table 1: Times in milliseconds t_{serial}, t_{mc}, and t_{gpu} with number of tracks N and total number of cameras C, where S_{mc} and S_{gpu} show the speedup on a multi-core CPU and a GPU compared to a serial implementation. In the two rightmost columns, ϵ and ϵ_L compare the average reprojection error in pixels against that of linear triangulation.

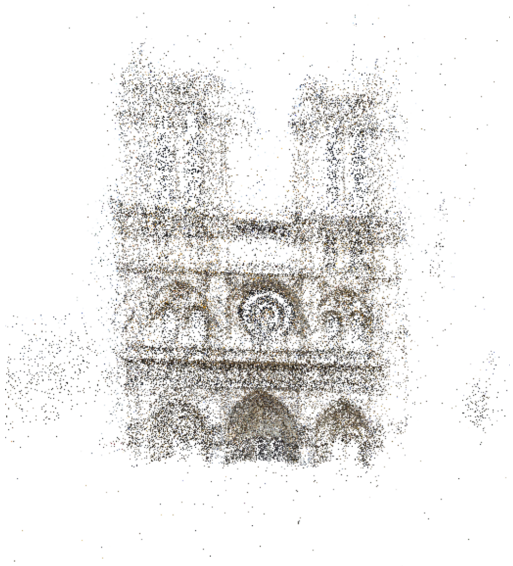


(a)



(b)

Figure 4: (a) GPU performance with increasing track error for different camera configurations. (b) Comparison of the one-thread-per-track and one-block-per-track implementations on a 20,000-track dataset and a 100,000-track dataset. Scaling is measured as track length is increased.



(a) *Notre Dame* [23] (290 views)

(b) *Brown12* [19] (337 views) (c) *ET* [23] (7 views)

Figure 5: Scenes reconstructed with our GPU triangulator.

5. Conclusion

In this paper, a fast and accurate GPU N-view triangulator is presented. An L_1-based cost function, which has

been shown to provide more accurate results for triangulation, maps well to the data-parallel model of the GPU when combined with statistical sampling. We develop and compare two parallelization approaches for the GPU, including the use of one thread to process one track, and also one thread block to process one track. We show that the performance of each approach depends on the number of tracks and track lengths in the datasets, providing flexibility in how it is used given different input data. The GPU triangulator is proven to be as accurate as the state of the art, while being several times faster than a serial implementation on real data and orders of magnitude faster than optimal algorithms. Our triangulator is designed to target large-scale reconstruction with ever-increasing image sizes and quantities, and opens the door for very accurate and efficient performance.

References

[1] The design and implementation of a generic sparse bundle adjustment software package based on the Levenberg-Marquardt algorithm. Technical Report 340, Institute of Computer Science – FORTH, Heraklion, Crete, Greece, Aug. 2000. 1, 3

[2] Eigen. http://eigen.tuxfamily.org, 2013. 6

[3] S. Agarwal, M. K. Ch, F. Kahl, and S. Belongie. Practical global optimization for multiview geometry. In *Proceedings of the 9th European conference on Computer Vision*, pages 592–605, 2006. 1, 3

[4] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In W. W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 26, pages 359–371. Morgan Kaufmann, Oct. 2012. 5

[5] M. Byröd, K. Josephson, and K. Åström. Fast optimal three view triangulation. In *Proceedings of the 8th Asian Conference on Computer Vision*, ACCV'07, pages 549–559, Berlin, Heidelberg, 2007. Springer-Verlag. 1, 3

[6] W. G. Cochran. *Sampling Techniques, 3rd Edition*. John Wiley, 1977. 4

[7] Z. Dai, Y. Wu, F. Zhang, and H. 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, Berlin, Heidelberg, 2012. Springer-Verlag. 1, 2, 3

[8] M. Fischler and R. Bolles. 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*, pages 726–740, 1987. 4

[9] R. Hartley and F. Kahl. Optimal algorithms in multiview geometry. In *Proceedings of the 8th Asian conference on Computer Vision – Volume Part I*, ACCV'07, pages 13–34, Berlin, Heidelberg, 2007. Springer-Verlag. 1, 2, 3

[10] R. I. Hartley and P. Sturm. Triangulation. *Comput. Vis. Image Underst.*, 68(2):146–157, 1997. 1, 3

[11] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2nd edition, 2004. 1, 2

[12] 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. 1

[13] P. Lindstrom. Triangulation made easy. In *Proceedings of the 2010 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1554–1561, 2010. 1, 3

[14] Y. 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, pages 488–494, Berlin, Heidelberg, 2010. Springer-Verlag. 1, 3

[15] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, pages 40–53, Mar./Apr. 2008. 4

[16] NVIDIA. Cuda C programming guide, version 4.2, 2013. 2

[17] Oxford Visual Geometry Group. Multi-view and Oxford Colleges building reconstruction, Aug. 2009. 6, 7

[18] S. Recker, M. Hess-Flores, and K. I. Joy. Statistical angular error-based triangulation for efficient and accurate multiview scene reconstruction. In *Workshop on the Applications of Computer Vision (WACV)*, 2013. 1, 2, 3, 4, 7

[19] M. I. Restrepo, B. A. Mayer, A. O. Ulusoy, and J. L. Mundy. Characterization of 3-d volumetric probabilistic scenes for object recognition. *IEEE Journal of Selected Topics in Signal Processing*, 6:522–537, Sept. 2012. 7

[20] J. R. Sánchez, H. Álvarez, and D. 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, Berlin, Heidelberg, 2010. Springer-Verlag. 2, 3

[21] J. R. Sánchez, H. Álvarez, and D. Borro. GPU optimizer: A 3D reconstruction on the GPU using Monte Carlo simulations – how to get real time without sacrificing precision. In *Proceedings of the 2010 International Conference on Computer Vision Theory and Applications*, pages 443–446, 2010. 2, 3

[22] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009. 5

[23] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: exploring photo collections in 3d. *ACM Transactions on Graphics*, 25(3):835–846, July 2006. 7

[24] J. A. Snyman. *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*. Applied Optimization, Vol. 97. Springer-Verlag New York, Inc., second edition, 2005. 1, 2, 3

[25] H. Stewénius, F. Schaffalitzky, and D. Nistér. How hard is 3-view triangulation really? *Computer Vision, IEEE International Conference on*, 1:686–693, 2005. 1, 2, 3, 6

[26] C. Strecha, W. von Hansen, L. J. V. Gool, P. Fua, and U. Thoennessen. On benchmarking camera calibration and multi-view stereo for high resolution imagery. In *Proceedings of the 2008 IEEE Conference on Computer Vision and Pattern Recognition*, 2008. 2, 4