Scientific Research Publishing

Engineering, 2023, 15, 663-690
https://www.scirp.org/journal/eng
ISSN Online: 1947-394X
ISSN Print: 1947-3931

Improving Accuracy and Computational Burden of Bundle Adjustment Algorithm Using GPUs

Pranay R. Kommera, Suresh S. Muknahallipatna*, John E. McInroy

Department of Electrical Engineering and Computer Science, University of Wyoming, Laramie, Wyoming, USA
Email: pkommera@uwyo.edu, *sureshm@uwyo.edu

How to cite this paper: Kommera, P.R., Muknahallipatna, S.S. and McInroy, J.E. (2023) Improving Accuracy and Computational Burden of Bundle Adjustment Algorithm Using GPUs. *Engineering*, **15**, 663-690.
https://doi.org/10.4236/eng.2023.1510046

Received: September 25, 2023
Accepted: October 28, 2023
Published: October 31, 2023

Abstract

Bundle adjustment is a camera and point refinement technique in a 3D scene reconstruction pipeline. The camera parameters and the 3D points are refined by minimizing the difference between computed projection and observed projection of the image points formulated as a non-linear least-square problem. Levenberg-Marquardt method is used to solve the non-linear least-square problem. Solving the non-linear least-square problem is computationally expensive, proportional to the number of cameras, points, and projections. In this paper, we implement the Bundle Adjustment (BA) algorithm and analyze techniques to improve algorithmic performance by reducing the mean square error. We investigate using an additional radial distortion camera parameter in the BA algorithm and demonstrate better convergence of the mean square error. We also demonstrate the use of explicitly computed analytical derivatives. In addition, we implement the BA algorithm on GPUs using the CUDA parallel programming model to reduce the computational time burden of the BA algorithm. CUDA Streams, atomic operations, and cuBLAS library in the CUDA programming model are proposed, implemented, and demonstrated to improve the performance of the BA algorithm. Our implementation has demonstrated better convergence of the BA algorithm and achieved a speedup of up to 16× on the use of the BA algorithm on various datasets.

Keywords

Bundle Adjustment, Levenberg-Marquardt, Scene Reconstruction, Radial Distortion Coefficient, Explicit Jacobian, CUDA Optimization

1. Introduction

Image-based three-dimensional (3D) scene reconstruction is an important component of Computer Vision. The image-based 3D scene reconstruction aims to

recreate a 3D geometric scene using the scene's two-dimensional (2D) images. The 3D scene reconstruction systems are prominent in the film industry, gaming, city and street modeling, and augmented reality to obtain 3D geometric properties of the scene. The reconstruction systems are also used in various fields like medical imaging [1] to reconstruct human body anatomy and geosciences [2] to generate 3D models of objects of geological interest.

A general 3D scene reconstruction pipeline involves multiple stages such as feature extraction [3], feature matching [4], camera parameters and 3D points initialization [5], camera parameters and 3D points refinement [6], and dense reconstruction. Precise information about camera parameters and 3D points is vital for accurate calibration and 3D reconstruction. They provide information about the orientation, measurements of the objects in the scene, and the position and depth of the various objects. As a result, refining camera parameters and 3D points plays an important role in the 3D scene reconstruction.

In this paper, we work on the refinement stage by implementing the Bundle Adjustment (BA) algorithm. The BA algorithm is used to refine the camera parameters and/or the 3D points by minimizing the reprojection error formulated as the summation of the differences between the computed reprojection and the observed projection. The total number of camera parameters and 3D points involved in the BA algorithm depends on the number of cameras generating an image each, the number of camera parameters being refined, the number of 3D points, and their projections across all the images. The computational load in the BA algorithm is proportional to the number of images, the total number of camera parameters, the number of 3D points, and their projections across the images. In this paper, we implement the BA algorithm on datasets [7] available online. The algorithm performance is optimized by improving the algorithm's convergence by minimizing the reprojection error and reducing the total computational time.

BA algorithm involves minimization of the reprojection error and can be represented as a non-linear least-squares problem linearized by approximation of the reprojection error function to a first-order Taylor polynomial expansion. The minimization of the reprojection error can be achieved by solving the system of Linear equations using methods like Gauss-Newton [8], Gradient Descent [9], and Levenberg-Marquardt (LM) [10] [11]. The LM method, a combination of Gauss-Newton and Gradient Descent, is used prevalently since it converges better than using only Gauss-Newton or Gradient Descent. Using Cholesky factorization, the linear system can be solved to obtain exact solutions leading to the exact-step LM method [6]. Even though the system of linear equations is symmetric positive definite (SPD), the computational burden increases polynomial with the number of images, points, and projections. Since an exact solution of the linear system is not required for 3D scene reconstruction, the linear system can be solved using the iterative LM method known as the inexact-step LM method. The inexact-step LM method [7] computes approximate solutions using an iterative solver, such as Conjugate Gradient (CG) or precon-

ditioned Conjugate Gradient (PCG) methods. Many researchers [6] [7] have demonstrated that the exact-step LM method is ideal for smaller datasets, and the inexact-step LM method is ideal for larger datasets.

In this paper, we implement the BA algorithm and evaluate techniques to improve the algorithm's performance by improving the convergence of the reprojection error. Camera parameters like rotation vector, translation vector, focal length, radial distortion, and 3D points are widely used in camera calibration. In addition, researchers in the state-of-the-art PBA [12] implementation have employed the first radial distortion coefficient, whereas implementation in [13] has used both the first and second radial distortion coefficient. Adding a second radial distortion coefficient would increase the total number of computations. In this paper, we evaluate using the second radial distortion coefficient to minimize the reprojection error and analyze the tradeoff between improving the reprojection error and increasing the computational load.

In addition to using the second radial distortion coefficient, we evaluate the effect of explicit analytical derivatives [14] by manually computing the Jacobian. Explicitly computing the Jacobian and using additional radial distortion coefficient increases the total number of computations in the BA algorithm. The increase in computational cost can be addressed by implementing the input/output operations on central processing units (CPUs) and concurrent sections of the code on graphics processing units (GPUs) using different performance optimization techniques.

CPUs and GPUs are two different types of processors that are widely used for computational purposes. The basic building blocks of both the processors contain similar components like cores and memory but vary in the numerical and functional configuration of these components. Generally, CPUs contain a smaller number of cores compared to GPUs with a higher clock rate thereby executing instructions faster compared to GPUs. Whereas GPUs with a higher number of cores can execute more instructions concurrently thereby resulting in higher throughput. The pipelining in the CPUs is more complex resulting in better execution of input/output, branching and logical operations compared to GPU cores which mainly target simple mathematical operations. In addition, the current CPUs have higher memory size compared to GPUs, whereas GPUs contain higher memory bandwidth compared to CPUs.

Recent advances in the computational capabilities of CPUs and GPUs have made them feasible for high-performance computations. The complex circuitry in CPUs, limited cores, and higher clock rates make them suitable for branching and input/output operations computations. The lightweight nature of the computational cores in huge numbers on GPUs makes them ideal for large-scale concurrent computations. In recent years, the CPU and GPU hardware have improved significantly in terms of resources on the chips, their performance, and efficiency. In addition to the improvements in the hardware, the software aspects, like the compilers and libraries used to exploit the parallelism in the computational methods on the respective hardware, have also improved signifi-

cantly. These improvements in hardware and software can address the computational limitations of the current BA algorithm implementations. In this paper, we demonstrate the use of GPUs and their features to address the increasing computational cost from the proposed additional radial distortion coefficient and explicit analytical derivatives. This is achieved by executing the compute-intensive concurrent operations on GPU and input/output, logical and branching operations on CPU.

The rest of the paper is organized as follows. Section 2 talks about various implementations of the BA algorithm. Section 3 provides information about the BA algorithm, LM method, and their mathematical representation. Section 4 talks about the Jacobian computation in PBA, which uses explicit analytical derivatives, and provides information about the additional radial distortion parameters used in this paper. In addition, Section 4 also provides information about the CUDA implementation of the BA algorithm. Section 5 provides information about the datasets and the performance parameters used in the paper. Section 6 demonstrates the results and provides information about the performance of the implemented sequential and CUDA versions of the BA algorithm. Finally, the paper concludes by reiterating the purpose for improved accuracy and computation cost of the algorithm, summarizes the findings and puts forward the scope of future work in the conclusion and future work section.

2. Literature Review

Refinement of camera parameters and 3D points in [15] has taken an incremental approach by adding information from one image at a time to the BA algorithm. This incremental approach has resulted in a significantly higher computational time. Efforts in [5] involve the use of the exact-step or inexact-step LM method based on the problem size, unlike [15], which used an incremental approach, a minimal subset of images that capture the dense connectivity [16], and the geometry of the scene are used to solve the BA algorithm.

The implementation in [6], which involves Cholesky decomposition and Schur complement, has proven to be accurate for solving linear systems using the LM method. With the increase in number of cameras and points, the sparsity has increased significantly in the linear systems, and the Schur complement with Cholesky decomposition is found to be computationally expensive. On the other hand, using the inexact-step LM method with the PCG method [7] [17] is appropriate for problems with a larger number of images. The convergence rate of the PCG method is dependent on the preconditioners used. Different preconditioners are studied [7] [17] [18] for the bundle adjustment problem.

BAL implementation [7] has compared the performance of the exact-step LM method with the inexact-step LM method using Jacobi, block Jacobi, and SSOR preconditioners. Multiscale preconditioning [18] was also proposed, improving the PCG method's convergence rate. In addition, researchers [19] have also implemented PCG methods for large-scale BA using a reduced camera system. The linear system is decomposed into blocks, and variable ordering is analyzed in

this work. They have compared the LDL factorization and PCG method performance with preconditioners confined to Jacobi and SSOR preconditioner. Ceres Solver [20], a C++ library to solve non-linear Least Squares problems, and Sparse Bundle Adjustment (SBA) [6] library is developed to compute the BA algorithm. Both these libraries take advantage of the sparse structure in the BA problems by implementing Block Compressed Sparse Row (BCSR) and Compressed Sparse Row (CSR) formats. Ceres Solver provides an analytic, numeric, and automatic derivatives interface.

Irrespective of the preconditioner used in the PCG method, sparse matrix–vector multiplication is one of the computationally expensive mathematical operations in the PCG method. The PBA implementation has tabulated the percentage of time spent on different operations. The time taken for matrix-vector multiplications is more than 80% of the time in the LM method. As a result, sparse matrix-vector multiplications play a significant role in the computational load of the PCG method.

A few efforts [12] [13] [21] were also made to implement the BA algorithm on many-core GPUs. Researchers [21] have developed a GPU version of the exact-step LM method using Compressed Column Storage (CCS). The computationally intensive linear systems are solved using the MAGMA library [22]. A GPU version of the inexact-step LM method using a block Jacobi preconditioner is developed in Parallel Bundle Adjustment (PBA) [12]. In the PBA implementation, only eight camera parameters are refined, consisting of three rotational elements, three translational elements, one focal length, and one radial distortion parameter. The BCSR format stores the sparse matrix, and the implementation has shown a significant performance boost compared to single-core execution. In this implementation, the augmented matrix is not computed; the Jacobian matrix is used directly in the computations. Researchers have utilized the cross-product between the partial derivatives of the 2D image vector with respect to the 3D point and partial derivatives of the 3D point with respect to axis-angle representation to compute the partial derivatives of the projection function with respect to the rotational vector. Approximations are employed to simplify the derivatives, resulting in zero values for a few derivatives. The PBA implementation can execute the code with and without the use of the Schur complement.

In the implementation [13], 11 camera parameters are refined, and only the time-consuming computations are simulated on GPU, and the remaining computations are simulated on CPU. Researchers have demonstrated their implementation to be better compared to the PBA implementation but have not provided a comparative analysis for their better convergence. In addition, the GPU performance is evaluated on datasets with fewer points and projections.

The current research efforts have targeted refining fewer camera parameters, parallelizing only the compute-intensive computations on the GPU, and working on datasets with fewer points and projections. In this research, we evaluate using second radial distortion in the camera parameters and explicit Jacobian

computation. Unlike [13], a comparative analysis of using the second radial distortion coefficient, its effect on the computational cost, and the algorithm convergence is presented. The convergence rate using the approximated Jacobian computation from the state-of-the-art implementation is compared with the exact Jacobian computations obtained from the proposed explicit derivatives. In the proposed framework, apart from parallelizing only the compute-intensive computation on GPU, all the computations in the LM iterations are ported to GPU, thereby eliminating intermediate data transfers. In addition, the algorithm's convergence rate and computation time are evaluated on datasets with more points and projections.

3. Bundle Adjustment Algorithm Formulation

3.1. Projection Function

The formulation of the BA algorithm depends on the 3D points being projected into the 2D images. The 3D world coordinates are mapped to the 2D camera pixel coordinates using a projection function that uses a pinhole camera model. Mathematically, a projection function [7] uses 3D point coordinates and camera parameters like rotation matrix, translation vector, focal length, and radial distortions to compute the 2D projection. The projection function formulated is shown in Equation (1).

$$f(\boldsymbol{P}) = f * \boldsymbol{r}(\boldsymbol{I}) * \boldsymbol{I} \tag{1}$$

$$\boldsymbol{P} = [\boldsymbol{R}, \boldsymbol{T}, f, K_1, K_2, \boldsymbol{X}]$$

$$\boldsymbol{r}(\boldsymbol{I}) = 1.0 + K_1 * \|\boldsymbol{I}\|^2 + K_2 * \|\boldsymbol{I}\|^4$$

where,

$f(\boldsymbol{P})$ is the projection function in 2D pixel coordinates

f is the focal length of the camera

$\boldsymbol{r}(\boldsymbol{I})$ is a scaling factor in terms of first and second radial distortion coefficients K_1 and K_2

$\boldsymbol{I} = [I_x, I_y] = [-O_x/O_z, -O_y/O_z]$ is the homogenous vector in the image plane

$\boldsymbol{O}$ is the 3D point $[O_x, O_y, O_z] = \boldsymbol{RX} + \boldsymbol{T}$ in camera coordinates

$\boldsymbol{R}$ is a 3 × 3 rotation matrix

$\boldsymbol{T}$ is a 3D translation vector

$\boldsymbol{X}$ is a 3D point

The scaling factor represented above in Equation (1) includes the second radial distortion coefficient K_2 which is analyzed in this work with respect to its effect on the convergence and the computational cost of the BA algorithm.

3.2. Bundle Adjustment Algorithm and Levenberg-Marquardt Method

The Bundle adjustment algorithm refines the 3D points and the camera parameters by minimizing the reprojection error, formulated as the summation of the

difference between the computed and observed projections. The minimization of the reprojection error for n points and m cameras can be formulated as shown in Equation (2).

$$\min \sum_{i=1}^{n} \sum_{j=1}^{m} \epsilon\left(\boldsymbol{P}, \boldsymbol{x}_{ij}\right) = \min \sum_{i=1}^{n} \sum_{j=1}^{m} \left\| \boldsymbol{x}_{ij} - f\left(\boldsymbol{P}\right) \right\|^2 \tag{2}$$

where,

$\epsilon\left(\boldsymbol{P}, \boldsymbol{x}_{ij}\right)$ is the projection error function

$\boldsymbol{x}_{ij}$ is the observed projection of point i in an image from the camera j

$f\left(\boldsymbol{P}\right)$ is the projection function used to obtain computed projections

In Equation (2), it can be observed that the reprojection error is a non-linear least-squares problem, and it can be solved using the Levenberg-Marquardt (LM) method. The LM method involves the approximation of a reprojection error with an updated parameter vector and produces a series of parameter vectors that minimize the reprojection error. As illustrated in [6] [7], the minimization results in a normal equation as in Equation (3).

$$\boldsymbol{J}^{\mathrm{T}} \boldsymbol{J} \boldsymbol{\delta} = -\boldsymbol{J}^{\mathrm{T}} \boldsymbol{e} \tag{3}$$

where,

$\boldsymbol{J} = \partial f\left(\boldsymbol{P}\right) / \partial \boldsymbol{P}$ is a Jacobian of the projection function

$\boldsymbol{\delta}$ is the change in the parameter vector

$\boldsymbol{e}$ is the error vector computed as the difference between the computed projection and observed projection

The Jacobian computed in the Equation (3) may result in a rank deficient matrix due to the limited information available from the camera-point system, resulting in the $\boldsymbol{J}^{\mathrm{T}} \boldsymbol{J}$ matrix to be a singular matrix. In such a case, solving Equation (3) will not guarantee convergence. As a result, to ensure convergence, a regularization term is added to Equation (3) resulting in an augmented normal equation shown below.

$$\left(\boldsymbol{J}^{\mathrm{T}} \boldsymbol{J} + \mu \boldsymbol{D}^{\mathrm{T}} \boldsymbol{D}\right) \boldsymbol{\delta} = -\boldsymbol{J}^{\mathrm{T}} \boldsymbol{e} \tag{4}$$

where,

$\boldsymbol{D}$ is a non-negative diagonal matrix formulated as the square root of the diagonal of the matrix $\boldsymbol{J}^{\mathrm{T}} \boldsymbol{J}$ [1]

μ is the positive damping parameter used to control the regularization

The augmented normal equation in Equation (4) can be represented as a linear system of equations as below.

$$\boldsymbol{A} \boldsymbol{x} = \boldsymbol{b} \tag{5}$$

where,

$\boldsymbol{A} = \boldsymbol{J}^{\mathrm{T}} \boldsymbol{J} + \mu \boldsymbol{D}^{\mathrm{T}} \boldsymbol{D}$ is a symmetric positive-definite matrix

$\boldsymbol{b} = -\boldsymbol{J}^{\mathrm{T}} \boldsymbol{e}$ is a gradient vector

$\boldsymbol{x} = \boldsymbol{\delta}$ is a solution vector

Using the regularization term, the matrix $\left(\boldsymbol{J}^{\mathrm{T}} \boldsymbol{J} + \mu \boldsymbol{D}^{\mathrm{T}} \boldsymbol{D}\right)$ becomes non-singular and positive definite ensuring convergence. The damping parameter controls the regularization's magnitude, allowing to alternate between the gra-

dient descent and the Gauss-Newton method based on the minima. A higher value of damping term results in the algorithm behaving as a gradient-descent algorithm thereby increasing the reduction in the residual. Lower values of damping term result in the algorithm to behave as a Gauss-Newton algorithm thereby slowing the reduction in the residual. The value of the damping parameter can be updated systematically [23] resulting in convergence to a better local minimum.

3.3. Mathematical Representation

The augmented normal equation in Equation (4) depends on the Jacobian and the error vector derived from the total number of cameras, 3D points, and their projections. Furthermore, it depends on the total number of optimized camera parameters. For example, consider three points projected onto two images. The computed projection vector and the parameter vector can be represented as shown below:

$$f\left(\boldsymbol{P}\right)=\left(\hat{\boldsymbol{x}}_{11}^{\mathrm{T}},\hat{\boldsymbol{x}}_{12}^{\mathrm{T}},\hat{\boldsymbol{x}}_{21}^{\mathrm{T}},\hat{\boldsymbol{x}}_{22}^{\mathrm{T}},\hat{\boldsymbol{x}}_{31}^{\mathrm{T}},\hat{\boldsymbol{x}}_{32}^{\mathrm{T}}\right)^{\mathrm{T}};\ \boldsymbol{P}=\left(\boldsymbol{c}_{1}^{\mathrm{T}},\boldsymbol{c}_{2}^{\mathrm{T}},\boldsymbol{p}_{1}^{\mathrm{T}},\boldsymbol{p}_{2}^{\mathrm{T}},\boldsymbol{p}_{3}^{\mathrm{T}}\right)^{\mathrm{T}} \tag{6}$$

where,

$\hat{\boldsymbol{x}}_{ij}^{\mathrm{T}}$ is the 2D computed projection of point i in camera j

$\boldsymbol{c}_{j}^{\mathrm{T}}$ and $\boldsymbol{p}_{i}^{\mathrm{T}}$ are the n-dimensional camera parameter vector of camera j and m dimensional point vector of point i

n and m are the total number of camera parameters and points, respectively

The sparse Jacobian matrix computed from the above projection and parameter vector can be represented as illustrated in [6]. Based on the Jacobian representation [6], all the notations [12] in Equation (4) can be categorized into camera and point sections as shown below:

$$\boldsymbol{J}=\left[\boldsymbol{J}_{c}\ \boldsymbol{J}_{p}\right];\ \boldsymbol{D}=\left[\boldsymbol{D}_{c}\ \boldsymbol{D}_{p}\right];\ \boldsymbol{\delta}=\left[\boldsymbol{\delta}_{c},\boldsymbol{\delta}_{p}\right];\ \boldsymbol{e}=\left[\boldsymbol{e}_{c},\boldsymbol{e}_{p}\right];\ \boldsymbol{\epsilon}=\left[\boldsymbol{\epsilon}_{c},\boldsymbol{\epsilon}_{p}\right] \tag{7}$$

The augmented normal equation in Equation (4) can be represented by the camera and point sections as below:

$$\begin{bmatrix}\boldsymbol{U}_{\mu} & \boldsymbol{W}\\ \boldsymbol{W}^{\mathrm{T}} & \boldsymbol{V}_{\mu}\end{bmatrix}\begin{bmatrix}\boldsymbol{\delta}_{c}\\ \boldsymbol{\delta}_{p}\end{bmatrix}=\begin{bmatrix}\boldsymbol{\epsilon}_{c}\\ \boldsymbol{\epsilon}_{p}\end{bmatrix} \tag{8}$$

$$\boldsymbol{U}_{\mu}=\boldsymbol{J}_{c}^{\mathrm{T}}\boldsymbol{J}_{c}+\mu\boldsymbol{D}_{c}^{\mathrm{T}}\boldsymbol{D}_{c};\boldsymbol{V}_{\mu}=\boldsymbol{J}_{p}^{\mathrm{T}}\boldsymbol{J}_{p}+\mu\boldsymbol{D}_{p}^{\mathrm{T}}\boldsymbol{D}_{p};\boldsymbol{W}=\boldsymbol{J}_{c}^{\mathrm{T}}J_{p};\boldsymbol{\epsilon}_{c}=-\boldsymbol{J}_{c}^{\mathrm{T}}\boldsymbol{e}_{c};\boldsymbol{\epsilon}_{p}=-\boldsymbol{J}_{p}^{\mathrm{T}}\boldsymbol{e}_{p}$$

Solving the augmented normal equation represented in Equation (8) using the PCG method involves computing the solution for the entire size of the solution vector. These computations can be reduced using the Schur complement, which involves multiplying both sides of Equation (8) with a block matrix as illustrated in [6] and represented in Equation (9) and Equation (10).

$$\left(\boldsymbol{U}-\boldsymbol{W}\boldsymbol{V}_{\mu}^{-1}\boldsymbol{W}^{\mathrm{T}}\right)\boldsymbol{\delta}_{c}=\boldsymbol{\epsilon}_{c}-\boldsymbol{W}\boldsymbol{V}_{\mu}^{-1}\boldsymbol{\epsilon}_{p} \tag{9}$$

$$\boldsymbol{V}_{\mu}\boldsymbol{\delta}_{p}=\boldsymbol{\epsilon}_{p}-\boldsymbol{W}^{\mathrm{T}}\boldsymbol{\delta}_{c} \tag{10}$$

Ongoing in this paper, solving Equation (8) directly for the solution vector is called *without-Schur* complement, and solving for the solution vector using the

Schur complement representation in Equation (9) and Equation (10) is called *with-Schur* complement.

4. Implementation

This section discusses the implementations proposed in this paper to improve the algorithm's convergence and reduce the time taken for the algorithm execution. The implementations are categorized into sequential implementation and CUDA implementation. All the implementations are developed and analyzed for the *without-Schur* and *with-Schur* complements.

4.1. Sequential Implementation

The performance in terms of convergence of the BA algorithm is analyzed using an additional radial distortion coefficient and explicitly computing the Jacobian matrix.

4.1.1. Additional Radial Distortion Coefficient

The camera parameters and points are primarily used to compute the reprojection error using the projection function and to compute the Jacobian matrix from the partial derivatives, which are then used to generate the augmented normal equation. In the PBA implementation, the camera parameters include an utmost of eight parameters, including three translation elements, three rotational elements in Rodrigue's vector representation, one focal length, and one radial distortion. The camera center is assumed to be at the origin, and the skew factor is set to 1. As always, the point vector includes three elements that provide the point's location in the 3D coordinates.

As a result, the size of each element-block [6] in the camera section of the Jacobian is (2 × 8) and the point section is (2 × 3). The size of the symmetric positive-definite matrix $\boldsymbol{A}$ is ((8 × number of cameras) + (3 × number of points))^2 and the size of both the gradient and solution vectors are ((8 × number of cameras) + (3 × number of points)).

In this paper, we analyze the effect of using the second radial distortion coefficient. As a result, a total of nine camera parameters, which include three translation elements, three rotation matrix elements in the form of Rodrigue's vector, one focal length, and two radial distortions, are used. Using the second radial distortion coefficient improves image correction compared to using only the first radial distortion coefficient. In addition, using an additional radial distortion coefficient will add additional constraints to the minimization problem in Equation (2), resulting in better refinement of the 3D points and camera parameters.

The projection function with the additional radial distortion coefficient is represented in Equation (1). The primary difference is in the scaling factor where the additional radial distortion parameter K_2 is implemented. The third radial distortion coefficient is not used in this implementation as it is a very small number in the order of less than −20 and does not significantly impact the

image correction.

The change in the number of camera parameters also impacts the size of each Jacobian and the augmented normal equation. The use of an additional radial distortion parameter (represented in bold) increases the size of the camera element block in the Jacobian, as shown in Equation (11).

$$\frac{\partial f\left(\boldsymbol{P}_{ij}\right)}{\partial \boldsymbol{c}^{j}}=\begin{bmatrix}\frac{\partial x_{ij}}{\partial c^{j}}\\ \frac{\partial y_{ij}}{\partial c^{j}}\end{bmatrix}=\begin{bmatrix}\frac{\partial x_{ij}}{\partial k_1^{j}} & \frac{\partial x_{ij}}{\partial k_2^{j}} & \frac{\partial x_{ij}}{\partial k_3^{j}} & \frac{\partial x_{ij}}{\partial t_1^{j}} & \frac{\partial x_{ij}}{\partial t_2^{j}} & \frac{\partial x_{ij}}{\partial t_3^{j}} & \frac{\partial x_{ij}}{\partial f^{j}} & \frac{\partial x_{ij}}{\partial K_1^{j}} & \frac{\partial \boldsymbol{x_{ij}}}{\partial \boldsymbol{K_2^{j}}}\\ \frac{\partial y_{ij}}{\partial k_1^{j}} & \frac{\partial y_{ij}}{\partial k_2^{j}} & \frac{\partial y_{ij}}{\partial k_3^{j}} & \frac{\partial y_{ij}}{\partial t_1^{j}} & \frac{\partial y_{ij}}{\partial t_2^{j}} & \frac{\partial y_{ij}}{\partial t_3^{j}} & \frac{\partial y_{ij}}{\partial f^{j}} & \frac{\partial y_{ij}}{\partial K_1^{j}} & \frac{\partial \boldsymbol{y_{ij}}}{\partial \boldsymbol{K_2^{j}}}\end{bmatrix} \tag{11}$$

where,

$\left(x_{ij}, y_{ij}\right)$ is the projection in camera system for point i and camera j

c^{j} is the camera parameter vector of camera j

$k^{j}=\left(k_1^{j}, k_2^{j}, k_3^{j}\right)$ is the rotation matrix of camera j in axis-angle representation

$t^{j}=\left(t_1^{j}, t_2^{j}, t_3^{j}\right)$ is the translation vector of camera j

f^{j} is the focal length of camera j

K_1^{j} and K_2^{j} are the first and second radial distortion coefficients of camera j

With the increase in element-block size in the Jacobian, every matrix/vector representation related to the camera also increases, as in Table 1.

4.1.2. Explicit Jacobian Computation

Each Jacobian element-block is computed as illustrated in [6] using Equation (3) and Equation (7). With nine camera parameters and three-point coordinates, the Jacobian element-blocks for each camera and point can be represented as in Equation (11) and Equation (12).

$$\frac{\partial f\left(\boldsymbol{P}_{ij}\right)}{\partial \boldsymbol{p}^{i}}=\begin{bmatrix}\frac{\partial x_{ij}}{\partial p^{i}}\\ \frac{\partial y_{ij}}{\partial p^{i}}\end{bmatrix}=\begin{bmatrix}\frac{\partial x_{ij}}{\partial x^{i}} & \frac{\partial x_{ij}}{\partial y^{i}} & \frac{\partial x_{ij}}{\partial z^{i}}\\ \frac{\partial y_{ij}}{\partial x^{i}} & \frac{\partial y_{ij}}{\partial y^{i}} & \frac{\partial y_{ij}}{\partial z^{i}}\end{bmatrix} \tag{12}$$

where,

Table 1. Size of different matrix/vector representations.

Size of	With 8 Camera Parameters rows × cloumns	With 9 Camera Parameters rows × cloumns
$\boldsymbol{J}_{c}^{ij}$	2 × 8	2 × 9
$\boldsymbol{U}_{j}$	8 × 8	9 × 9
$\boldsymbol{A}$	$\left(\left(8\times \mathrm{num}_{\mathrm{cam}}\right)+\left(3\times \mathrm{num}_{\mathrm{pt}}\right)\right)^{2}$	$\left(\left(9\times \mathrm{num}_{\mathrm{cam}}\right)+\left(3\times \mathrm{num}_{\mathrm{pt}}\right)\right)^{2}$
$\boldsymbol{\delta}$	$\left(\left(8\times \mathrm{num}_{\mathrm{cam}}\right)+\left(3\times \mathrm{num}_{\mathrm{pt}}\right)\right)\times 1$	$\left(\left(9\times \mathrm{num}_{\mathrm{cam}}\right)+\left(3\times \mathrm{num}_{\mathrm{pt}}\right)\right)\times 1$
$\boldsymbol{\epsilon}$	$\left(\left(8\times \mathrm{num}_{\mathrm{cam}}\right)+\left(3\times \mathrm{num}_{\mathrm{pt}}\right)\right)\times 1$	$\left(\left(9\times \mathrm{num}_{\mathrm{cam}}\right)+\left(3\times \mathrm{num}_{\mathrm{pt}}\right)\right)\times 1$

p^i is the point parameter vector point i

$\left(x^i, y^i, z^i\right)$ are the 3D coordinates of point i

Computing the partial derivatives concerning translation vector, focal length, and radial distortion are straightforward. But the partial derivatives concerning rotation vector in axis-angle representation is implemented [12] [24] through the cross product of the partial derivatives of the 2D image vector with respect to 3D point, and partial derivatives of 3D point with respect to the axis-angle representation as below.

$$\frac{\partial P}{\partial X} = \begin{bmatrix} \frac{f}{z} & 0 & -\frac{fx}{z^2} \\ 0 & \frac{f}{z} & -\frac{fy}{z^2} \end{bmatrix} \quad \frac{\partial X}{\partial w} = \begin{bmatrix} 0 & -k_3 & k_2 \\ k_3 & 0 & -k_1 \\ -k_2 & k_1 & 0 \end{bmatrix} \tag{13}$$

where,

P is the projection computed

$X = \left(x, y, z\right)$ is the 3D point in camera coordinates computed from rotation matrix R derived using Rodrigues' formula from its axis-angle representation

f is the focal length

$w = \left(k_1, k_2, k_3\right)$ is the rotational vector representation in axis-angle format

The rotation matrix is computed from the axis-angle representation using Rodrigues' formula [25], and the Jacobian for rotation vector in axis-angle representation is computed as shown in Equation (13). As a result, when the parameters are updated after a successful LM iteration, the updated rotation matrix is computed as Equation (14).

$$R' = dR * R_{ori}^{\mathrm{T}} \tag{14}$$

where,

R' is the updated rotation matrix

R_{ori}^{T} is the transpose of the original rotation matrix

dR is the change in rotation matrix. For simplicity, the above equation does not include radial distortion parameter

PBA implementation has adapted the above methodology and the partial derivative of y_{ij} and x_{ij} with respect to translation elements t_1 and t_2 are approximated to 0, although they contain smaller non-zero values.

Unlike in PBA implementation, we propose using a rotation vector in the axis-angle representation directly in the Jacobian computation without computing the rotational matrix using Rodrigues' formula and without any cross-product of partial derivatives. The Jacobian is derived using explicit analytical derivatives. By using rotation vector in axis-angle rotation, there is no need to update the rotation matrix as in Equation (14); instead update the rotation matrix in the axis-angle representation as shown in Equation (15).

$$k' = k^{ori} + \delta_k \tag{15}$$

where,

$k' = \left[k_1' \, k_2' \, k_3'\right]$ are the update rotation matrix in axis-angle representation

$k^{ori} = \left[k_1^{ori} \; k_2^{ori} \; k_3^{ori} \right]$ are the original rotation matrix in axis-angle representation

$\delta_k = \left[\delta_{k_1} \; \delta_{k_2} \; \delta_{k_3} \right]$ are the change in the rotation matrix in axis-angle representation

Unlike in PBA implementation, the partial derivative of y_{ij} and x_{ij} with respect to translation elements t_1 and t_2 are not approximated to 0. The expressions for the partial derivates are complex to be derived manually. As a result, the derivates of the Jacobian elements are computed using "diff" command in Matlab Symbolic Math Toolbox [26] and by hardcoding the projection function into the code.

Using explicit Jacobian matrix and the additional radial distortion coefficient increases the total computations. This increase in the computational load is addressed by implementing the code using the CUDA programming model on the GPUs.

4.2. CUDA Implementation

As the Jacobians of each projection function depend on one set of camera and 3D points only, most elements of the Jacobian matrix would end up being zero, resulting in a sparse matrix as detailed in [6]. Computing normal equations from the Jacobian matrix and solving the LM implementation would involve numerous matrix and vector operations that benefit heavily from the cache. Also, the independent nature of many computations in the BA algorithm would benefit from the asynchronous computations. The features as mentioned earlier, and the behavior of the BA algorithm would benefit from simulating different code sections concurrently on the GPUs. As a result, the Compute Unified Device Architecture (CUDA) [27] application programming interface (API) is used to accelerate the BA algorithm using the inexact-step LM method with the PCG method on GPUs.

This paper implements the BA algorithm on GPUs using the CUDA programming model. The proposed explicit Jacobian and an additional radial distortion coefficient in the sequential implementation are also used in the CUDA implementation. Implementing the BA algorithm on the GPUs using CUDA involves many aspects of memory transfers and block-thread configuration on the GPUs for optimal concurrency. This paper implements basic optimizations involved in CUDA programming for effective memory transfers and computational throughput.

In this paper, we study the use of different CUDA features/libraries to improve the computational performance of the algorithm compared to its sequential implementation. A highly optimized CUDA BLAS library, cuBLAS [28], is used for all the vector operations. CUDA streams are used to take advantage of the asynchronous behavior of the mathematical equations in the algorithm. In addition, owing to the sparse structure of the matrix, Atomic Additions are used in the matrix-vector multiplications, thereby taking advantage of the concurrency from the parallel computations, and making sure to eliminate any race

condition from writing to the same memory. Parallelizing compute-intensive computations on the GPU and transferring the data to the CPU is ideal for smaller datasets, but the data transfer penalty would be significant for huge datasets. The proposed framework is developed to execute the entire BA algorithm on the GPUs, eliminating intermediate data transfers of considerable sizes to the CPU.

4.2.1. Computational and Memory Tradeoff

Solving the augmented normal equation requires the computation of matrices represented on the left-hand side of Equation (8). In the PBA implementation, the matrix is not computed explicitly but through the matrix-vector product [12] in the PCG method. The PCG method is implemented using a block Jacobi preconditioner. Even though the matrix is not explicitly computed, the diagonal blocks in matrix $\boldsymbol{A}$ need to be computed for the sake of the preconditioner. This would involve multiple computations of block diagonal matrices $\boldsymbol{U}_\mu$ and $\boldsymbol{V}_\mu$.

Instead of computing block diagonals multiple times, we store their values in the memory, thereby reducing the additional computations. Storing the matrices does not require a significant memory as their sizes are smaller. In addition, the gradient vector is also computed and stored in the memory as it is required to check the termination conditions in the algorithm. Storing the required matrices and vectors instead of recursive computations has resulted in around 3% performance improvement in the computational time.

4.2.2. Concurrent Executions

In the BA algorithm, solving the augmented normal equation can be categorized into camera and point sections and be executed concurrently. The computation of block diagonal matrices and their inverse for the preconditioner in the PCG method are independent of each other. In addition, matrix-vector computations in the PCG method can also be executed concurrently per camera and point section. Concurrent executions of the camera and point sections will improve the algorithm's performance by reducing the computational time. The serial and concurrent execution of the matrix-vector multiplication in the PCG method is shown below:

$$\boldsymbol{A}p^k = \begin{bmatrix} \boldsymbol{U}_\mu & \boldsymbol{W} \\ \boldsymbol{W}^{\mathrm{T}} & \boldsymbol{V}_\mu \end{bmatrix} \begin{bmatrix} p_c^k \\ p_p^k \end{bmatrix} = \begin{bmatrix} \boldsymbol{U}_\mu p_c^k + \boldsymbol{W} p_p^k \\ \boldsymbol{W}^{\mathrm{T}} p_c^k + \boldsymbol{V}_\mu p_p^k \end{bmatrix} \tag{16}$$

From **Figure 1** it can be seen that the matrix-vector computations take around six steps when computed sequentially but take only two steps when computed concurrently. This paper uses CUDA streams [27] to implement concurrency in the algorithm. Multiple CUDA streams are created and used across the algorithm where the operations are independent. **Figure 2** and **Figure 3** provide an NSight Systems [29] profiler view of the inverse computation of the diagonal matrices with and without using streams.

The "*cudaUinv*" and "*cudaVinv*" kernels compute the inverse of block diagonals in the preconditioner computation. The kernels in **Figure 2** are executed

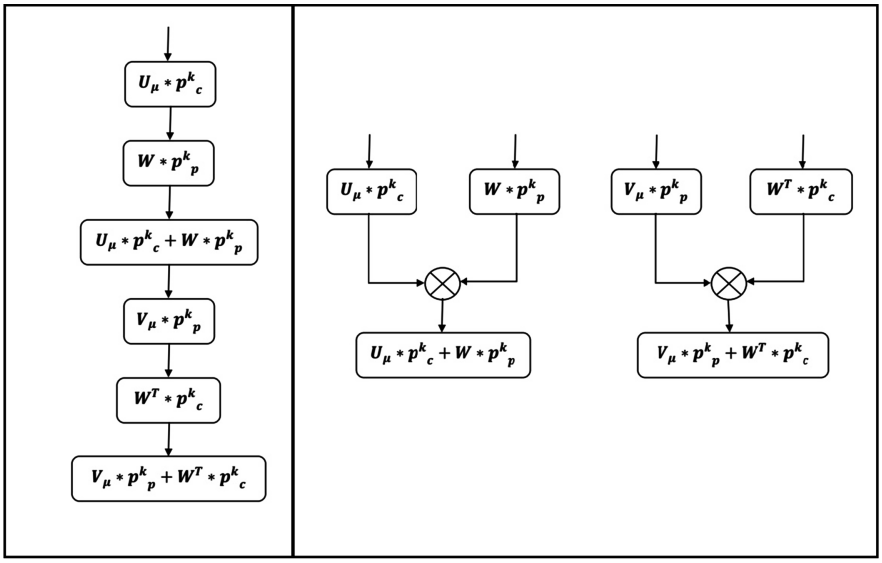

Figure 1. Sequential (left) and concurrent execution (right) of different stages in the matrix-vector multiplication involved in PCG method.

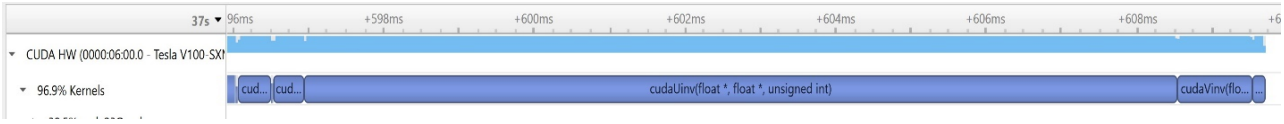

Figure 2. Profiler view of the inverse of preconditioner without CUDA streams.

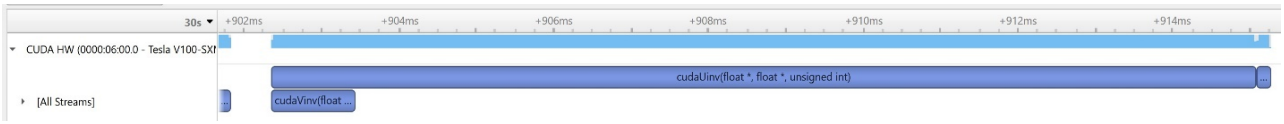

Figure 3. Profiler view of the inverse of the preconditioner with CUDA streams.

consecutively, one after the other, according to their calling. Whereas in **Figure 3**, with CUDA streams implementing the kernels on different streams, the kernel execution starts around the same time. As a result, the computational time taken by "*cudaVinv*" is hidden behind the computational time of the "*cudaUinv*" kernel, thereby reducing the execution time for the entire preconditioner computation. Similarly, various computations of camera and point sections are independent of each other, and their concurrency is exploited using the CUDA streams.

4.2.3. cuBLAS Implementation

Computation of the right-hand side of the normal equation involves matrix-matrix multiplications of Jacobians. As in PBA implementation [12], the matrix-matrix computations are replaced by the matrix-vector operation, and most of the computations in the BA algorithm now involve matrix-vector and vector-vector operations. cuBLAS is a highly optimized BLAS library containing APIs for matrix-vector and vector-vector operations, which can be used to implement code on GPUs. cuBLAS APIs like L^2-norm, dot product, vector addition, and scalar-vector multiplication are also utilized in the code, resulting in a

performance boost of around 5% - 10% compared to the manually optimized parallel kernels.

4.2.4. Atomic Operations

Computation of block diagonal matrices and matrix-vector multiplication operations involve sweeping through all the projections. PBA implementation [12] stores shuffled copies of data-in-order to enable improved continuous memory access patterns. In this paper, we implemented atomic operations [27] supported by the CUDA platform. The atomic operations ensure that the same memory location is not modified by any other thread, thereby preventing any read-after-write hazard. As the computations are iterated through the projections, we can distribute each computation concerning a projection onto one thread on GPU and use atomic operations to update the values appropriately.

4.2.5. Thread and Block Configuration on GPU

Different thread and block configurations would impact the performance of the CUDA kernels. Using very few threads in a block would result in underutilization of the compute resources. On the other hand, utilizing more threads might hit the resource limitations and spawn fewer blocks, resulting in less parallelism. As a result, the thread and block configurations should be set based on the computational load and the resources available per streaming multiprocessor on a GPU.

In the BA algorithm, most of the matrix-vector, vector-vector operations are iterated through several cameras, points, projections, and the size of the solution vector. As the number of cameras is fewer compared to points and projections, assigning computations of each camera to a block has produced optimal performance for CUDA kernels that iterate through the cameras. On the other hand, as the number of points and projections is higher, unrolling computations of multiple points/projections per block has reduced the total number of blocks invoked and produced optimal performance.

5. Datasets and Performance Parameters

5.1. Datasets

The datasets used in this paper are obtained from the online resources provided by Bundle Adjustment in the Large [7] (https://grail.cs.washington.edu/projects/bal/). The online resource provided has around 100 different datasets. Below is a table with information about a few different datasets whose final mean square error, computational time, and speedup are demonstrated in the paper. The table is arranged in increasing order of the total number of projections.

Of these total datasets, about 32 of them do not converge from their initial mean square error. This indicates that these datasets cannot be further refined. Apart from these datasets, some produce different behavior involving Not a Number (NaN) errors. In this paper, 66 different datasets are executed and analyzed. Apart from the datasets provided in **Table 2**, a summary of the behavior

Table 2. Details about the number of images, points, and projections for different datasets.

Dataset No.	Number of Images	Number of Points	Number of Projections
1	21	11,315	36,455
2	138	44,033	165,899
3	856	93,344	415,769
4	1587	150,845	663,289
5	287	182,023	971,292
6	308	195,089	1,045,197
7	356	226,730	1,255,268
8	961	187,103	1,692,975
9	871	527,480	2,785,977
10	1936	649,673	5,213,733

of the remaining datasets will also be presented in the paper using the performance parameters mentioned below.

5.2. Performance Parameters

The convergence of the mean square error of the algorithm is used as a measurement in the sequential implementations. Lowering the mean square error indicates better convergence. When comparing two different implementations percentage error between the final mean square errors is calculated below.

$$\delta = \frac{\text{mse}_{\text{original}} - \text{mse}_{\text{modified}}}{\text{mse}_{\text{original}}} \times 100\% \tag{17}$$

where,

δ is the percentage error

$\text{mse}_{\text{original}}$ is the final mean square error of the original code

$\text{mse}_{\text{modified}}$ is the final mean square error of the modified code

The aggregate performance of all the datasets will be computed using the average percentage error, as shown in Equation (18).

$$\text{Average Percentage Error} = \frac{\sum_{k=1}^{K} \delta^k}{K} \tag{18}$$

where,

δ^k is the percentage error of k^{th} dataset

K is the total number of datasets

The positive average percentage error indicates the percentage of improvement achieved by the modified code.

The speedup of the algorithm is used as a measurement to compare the performance boost achieved by the CUDA implementation compared to the sequential implementation. The speedup can be computed from Equation (19).

The higher the speedup, suggests better the performance of the modified implementation.

$$\text{Speedup} = \frac{T_{\text{base}}}{T_{\text{modified}}} \tag{19}$$

where,

T_{base} is the time taken by the base algorithm

T_{modified} is the time the modified algorithm takes

6. Results and Performance Analysis

The proposed framework and the PBA implementation are implemented on a server equipped with a 64-core AMD EPYC 7713P CPU and 3584 cores NVIDIA A30 GPU. The AMD CPU and NVIDIA GPU are connected using a PCIe interface with a bandwidth of 64GB/s. The proposed framework and the PBA library use the C/C++ programming language for the CPU implementation and the CUDA C programming model for the GPU implementation. In addition, the cuBLAS library [28] is used in the proposed framework to implement the CUDA version of basic linear algebra subroutines (BLAS).

First, the comparisons using different Jacobian computations and additional camera parameters between the sequential implementations of the proposed implementation and the PBA implementation are evaluated. Next, the CUDA version of the proposed framework is compared with the CPU version of the proposed framework and the CUDA version of the PBA library. All the comparisons involve both the algorithm configurations of *without-Schur* and *with-Schur*.

6.1. Sequential Implementation

6.1.1. Explicit Jacobian Computation

Explicitly computing the Jacobian affects the convergence of the algorithm. **Table 3** compares the convergence of different datasets from **Table 2** between the PBA implementation and the proposed explicit Jacobian implementation. Both the configurations of the algorithm *without-Schur* and *with-Schur* complements are provided.

Table 3 shows that for a higher number of datasets, the explicit Jacobian implementation convergence is better or similar compared to the PBA implementation. Eliminating the approximations in the Jacobian computation would impact the convergence. There are multiple datasets where the PBA implementation convergence is better than the explicit Jacobian implementation. The average percentage error for those *without-Schur* complement is 4.7%, and those *with-Schur* complement is 3.7%. The positive value of the average percentage error indicates that the algorithm has a better convergence using the explicit Jacobian implementation. Of the 66 datasets analyzed, 75% have better convergence using the explicit Jacobian implementation in the without-Schur configuration, and 83% have better convergence using the explicit Jacobian implementation in the *with-Schur* configuration.

Table 3. The final mean square error values across different datasets using *without-Schur* and *with-Schur* complement for both the PBA Jacobian and Explicit Jacobian implementations.

Dataset No.	Final Mean Square Error					
	Without-Schur Complement			*With-Schur* Complement		
	PBA	Explicit Jacobian	Rate of Change	PBA	Explicit Jacobian	Rate of Change
1	1.846	1.844	0.002	1.846	1.844	0.002
2	1.239	1.238	0.001	1.239	1.237	0.002
3	1.271	1.117	0.154	1.282	1.116	0.166
4	1.595	1.347	0.248	1.796	1.332	0.464
5	1.146	1.262	−0.116	1.163	1.157	0.006
6	1.408	1.414	−0.006	1.242	1.061	0.181
7	1.451	1.649	−0.198	1.412	1.354	0.058
8	1.967	1.968	−0.001	1.967	1.968	−0.001
9	1.380	1.383	−0.003	1.386	1.383	0.003
10	1.929	1.937	−0.008	1.929	1.937	−0.008

6.1.2. Additional Camera Parameters

The use of an additional radial distortion coefficient has a significant impact on the convergence of the algorithm. Table 4 shows the final mean square error across different datasets *without-Schur* and *with-Schur* complement. The explicit Jacobian implementation is employed using eight and nine camera parameters. Unlike in Table 3, the difference between the final mean square errors is higher. The convergence of the datasets is higher when nine camera parameters are used. The average percentage error for those *without-Schur* complement is 17%, and for those *with-Schur* complement is 16%.

Although additional camera parameters would increase the total number of computations and the memory usage, the algorithm's convergence is significantly improved. Using a second radial distortion coefficient results in additional dimension in the Jacobian elements and the projection vector, providing more information for the camera and point refinement. Of the entire 66 datasets analyzed, all the datasets have better convergence using nine camera parameters in the *without-Schur* configuration, and 98% of the datasets have better convergence using nine camera parameters in the *with-Schur* configuration.

6.1.3. Convergence Analysis

The minimization problem aims to converge the computed projections, which are evaluated using the camera parameters and 3D points to the observed projections. In an ideal case, the mean square error would result in zero, where both the observed and computed projections are similar. The convergence of the computed projections to the observed projections by refining the camera parameters and 3D points is analyzed using the L-infinity norm in Table 5.

Table 4. The final mean square error values across different datasets using *without-Schur* and *with-Schur* complement for the eight and nine camera parameters implementations.

Dataset No.	Final Mean Square Error					
	Without-Schur Complement			*With-Schur* Complement		
	8 Camera Parameters	9 Camera Parameters	Rate of Change	8 Camera Parameters	9 Camera Parameters	Rate of Change
1	1.844	1.667	0.177	1.844	1.667	0.177
2	1.238	1.065	0.173	1.237	1.045	0.192
3	1.117	1.021	0.096	1.116	1.059	0.057
4	1.347	1.175	0.172	1.332	1.159	0.173
5	1.262	0.723	0.539	1.157	0.724	0.433
6	1.414	0.743	0.671	1.061	0.743	0.318
7	1.649	0.784	0.865	1.354	0.783	0.571
8	1.968	1.874	0.094	1.968	1.874	0.094
9	1.383	1.236	0.147	1.383	1.237	0.146
10	1.937	1.784	0.153	1.937	1.784	0.153

Table 5. L-Infinity norm of the projection errors.

Dataset No.	L-Infinity Norm of the Projection Errors Represented in L-2 Norm				
	Initial	*Without-Schur* Complement		*With-Schur* Complement	
		PBA	Proposed	PBA	Proposed
1	0.01727	0.00478	0.00468	0.00478	0.00468
2	0.02075	0.00824	0.00815	0.00824	0.00816
3	2.39244	0.22943	0.02752	0.06158	0.13779
4	26.5631	0.15234	0.13674	0.46556	0.10041
5	0.11949	0.05866	0.00969	0.05787	0.00969
6	0.11345	0.06318	0.01152	0.05902	0.01151
7	0.11403	0.06488	0.01263	0.05927	0.01263
8	0.09239	0.04870	0.04862	0.04870	0.04862
9	0.04171	0.02299	0.02306	0.02300	0.02308
10	0.23783	0.05419	0.05418	0.05419	0.05418

First, each of the 2D projection errors computed initially is converted into a scalar value by using the L-2 norm on each projection, resulting in a 1D projection error vector. L-infinity norm is then applied to the 1D projection error vector to identify the projection index with the largest projection error. The 'Initial' column in Table 5 provides information on the L-infinity norm for the projec-

tion errors derived from the initial camera parameters and 3D points. Table 5 also provides information about the final L-Infinity norm for the projection errors derived from the refined camera parameters and 3D points.

Table 5 shows that the proposed implementation with a second radial distortion coefficient and the explicit Jacobian computation results in a lower L-infinity norm than the PBA implementation. The overall reduction of the projection error indicates that the proposed implementation improves the algorithm's convergence. The difference between the L-infinity values provided can be observed to be very small in the order of −1 to −4. This is because of the data being normalized. Denormalization of the data would provide a better understanding of the effect of the refined camera parameters and 3D points.

Table 6 provides the observed and computed projection in pixel coordinates after denormalization. A comparison between the observed projection to the final computed projection for the projection index with higher projection error from both the PBA and Proposed implementations is shown in Table 6. The comparison is done by computing the distance between the observed projection and the computed projection in terms of pixel coordinates.

Table 6 shows that the proposed implementation has better convergence and the final computed projection in terms of pixel coordinates is closer to the observed projection than the PBA implementation. Figure 4 provides a pictorial representation of the computed and observed projections for different projection indexes. Figure 4 shows that the computed projections from the proposed implementation are close to the observed projections compared to the computed projections from the PBA implementation.

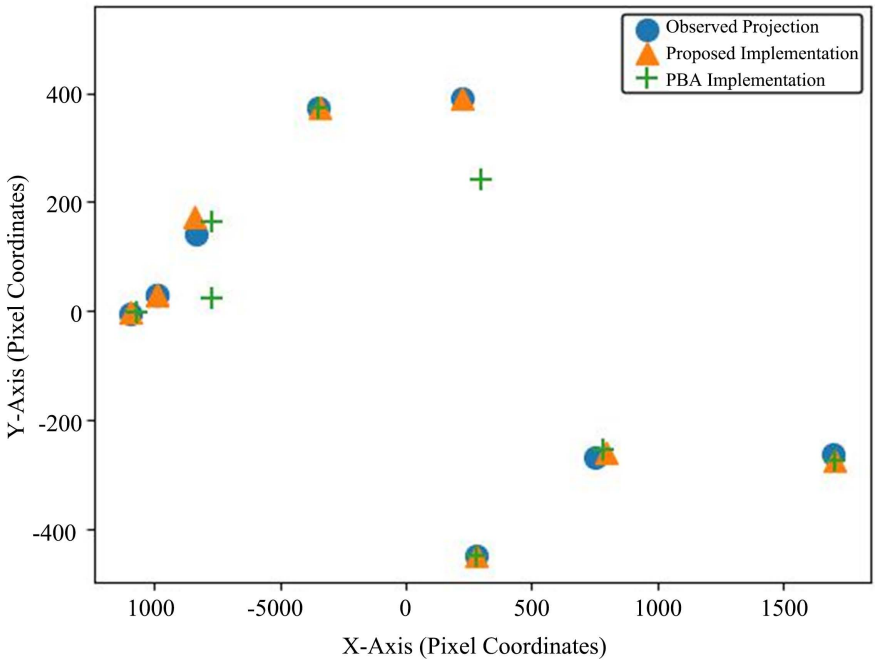

Figure 4. Final projections in pixel coordinates from both observed and computed projections.

Table 6. Final projections are represented in pixel coordinates.

Dataset No.	*Without-Schur* Complement		*With-Schur* Complement	
	PBA	Proposed	PBA	Proposed
1	9.0172	11.1571	9.0205	11.1583
2	2.0843	1.5112	2.0937	0.9540
3	0.4697	0.2444	0.9435	0.2673
4	147.0987	0.0450	152.1348	0.0016
5	68.4172	33.4208	65.4357	33.4225
6	217.9652	1.8467	203.614	1.8483
7	222.7412	1.8852	203.4508	1.8588
8	34.0829	45.6782	34.0821	45.9777
9	0.1618	0.5123	0.1626	0.4891
10	25.9884	4.5954	25.9916	4.6031

6.2. CUDA Implementation

Based on the sequential implementation analysis, the algorithm's CUDA version is analyzed using explicit Jacobian implementation and nine camera parameters. This paper implements a preliminary CUDA version of the algorithm resembling the sequential implementation with basic GPU-related optimizations. Unlike the PBA algorithm, which is heavily dependent on the texture memory deprecated in the latest CUDA software stack, we are confining ourselves to effectively utilize global, shared, and register memory. In addition, we have employed the highly optimized cuBLAS library for all the linear algebra computations. As the algorithm is highly sensitive to precision changes, similar behavior between the sequential and CUDA implementations which are executed on different hardware, is of utmost importance. As a result, we have initially compared the final mean square error between the sequential and CUDA implementations, followed by analyzing the speedup for different datasets.

The final mean square error is approximately similar across CPU and GPU implementations. The overall average percentage error for *without-Schur* complement is 0.05% and for *with-Schur* complement is 0.2%. Of the total 66 datasets, greater than 90% have the percentage error less than an order of −2. The lower average percentage error indicates that sequential and CUDA implementations converge to the same minima. Although both the implementations converge to the same minima, the number of LM steps and the total number of conjugate gradients iterations it takes to reach the minima is different, and it is mainly because of the differences in precision across hardware.

6.2.1. Computational Time and Speedup between CPU and GPU Implementations

Figure 5 shows the time taken by sequential and CUDA implementations across

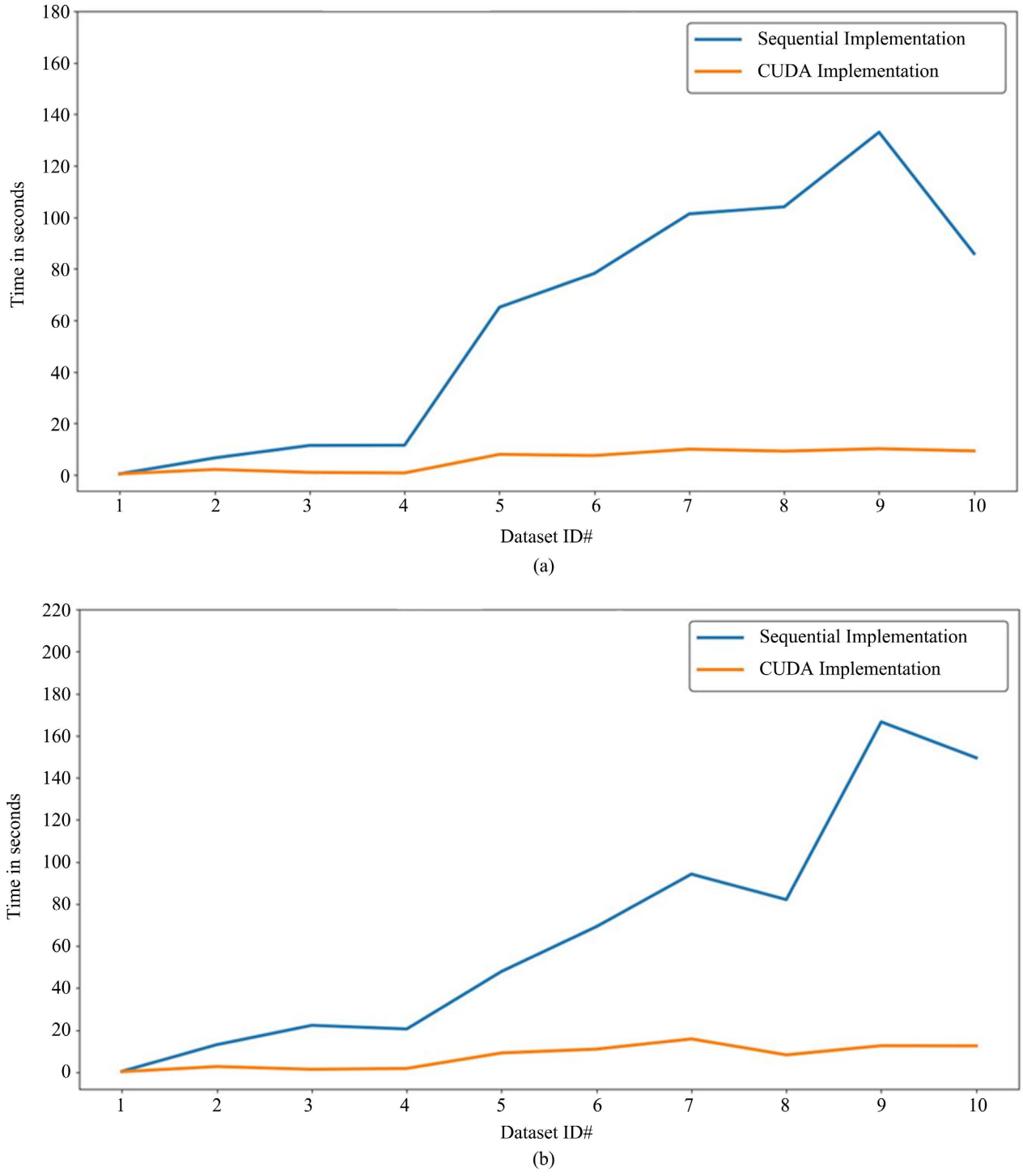

Figure 5. Time taken by sequential and CUDA implementation across different datasets using *without-Schur* and *with-Schur* complement. (a) *Without-Schur* Complement; (b) *With-Schur* Complement.

different datasets. The plot shows that the CUDA implementation always takes less time compared to the sequential implementation. The time taken by the *with-Schur* complement configuration is higher than the *without-Schur* complement as it involves additional computations of the Schur complement. The

time taken by a dataset depends upon many factors like the size of the delta vector, which is (9 × number of cameras) + (3 × number of points), number of projections, total LM iterations and total number of conjugate gradient iterations. From Figure 5, it can be seen that the total number of projections significantly impacts the time taken by the datasets.

The speedup between the sequential and CUDA implementations for the above datasets is provided in Figure 6. From the plot, it can be observed that the speedup is random across the datasets from 1.02× to 16×. As the time taken by the dataset depends on many factors, as mentioned earlier, a smaller dataset might take more time than a larger dataset if it requires more LM steps or conjugate gradient iterations. For example, the time taken by the Dataset ID# 9 is greater than the time taken by the Dataset ID#10. Although Dataset ID#10 has 5,213,733 projections compared to the 2,785,977 projections in Dataset ID# 9, the total number of conjugate gradient iterations taken by Dataset ID# 9 is around 1400 compared to the 600 conjugate gradient iterations taken by Dataset ID# 10. From our analysis, it is observed that the size of the dataset has a significant impact on the speedup compared to the total LM steps and the conjugate gradient iterations. The dataset's size depends on the number of cameras, points, and projections, of which the total number of projections has more impact on the speedup.

To demonstrate, we have obtained the speedup for different datasets with an increased number of projections by setting the total number of LM iterations and the conjugate gradient iterations to one, as shown in Figure 7. From the plot, it can be observed that the speedup increases as the total number of projections increases. But for a smaller number of projections, the total number of cameras and points also affects the speedup.

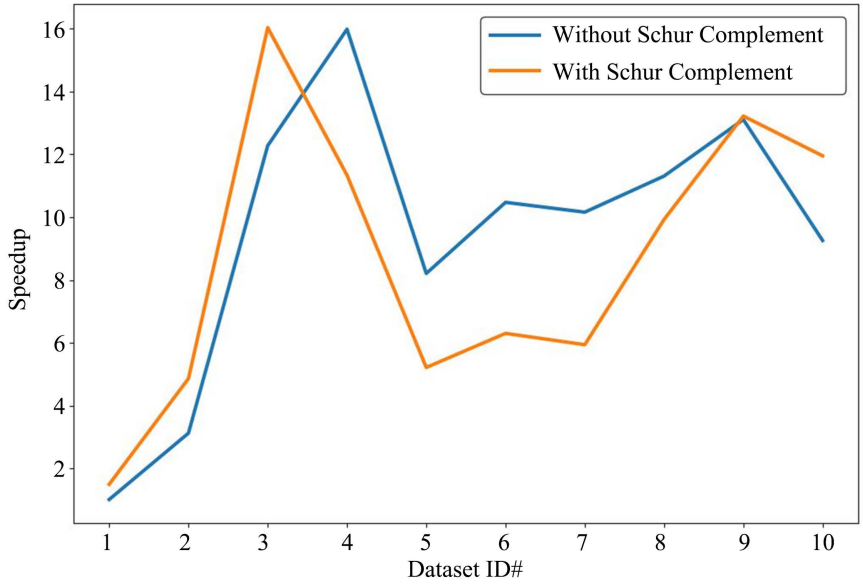

Figure 6. Speedup between the sequential and CUDA implementation using *with-Schur* and *without-Schur* complement across different datasets.

In addition to the speedup plot in Figure 7, Table 7 provides information about the time taken by sequential and CUDA implementations for eight and nine camera parameters using both *Without-Schur* and *With-Schur* complement. All the simulations are executed for one LM and one CG iteration. From the Table, it can be seen that the time taken for nine camera parameters is higher than for eight camera parameters for both the CPU and GPU implementations. This is because of the additional computations involved with the additional

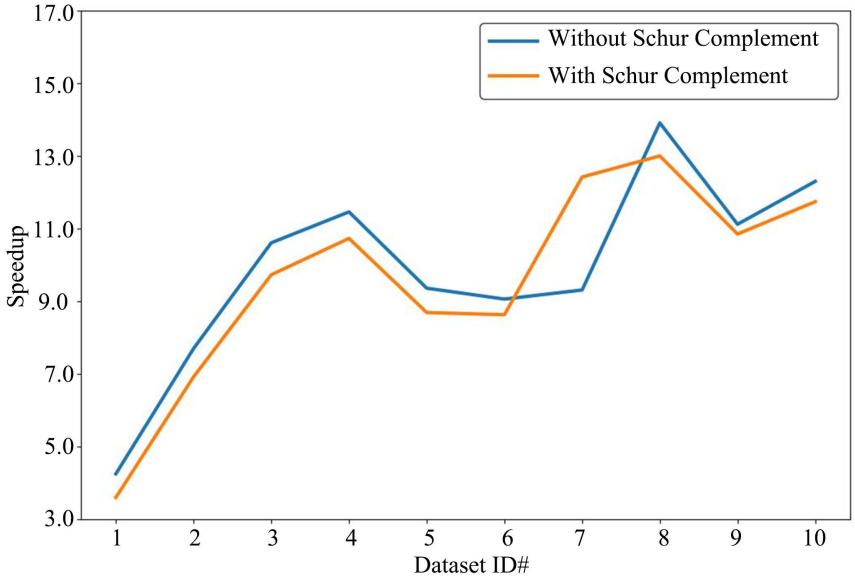

Figure 7. Speedup plot for different datasets using *without-Schur* and *with-Schur* complement with 1 Levenberg-Marquardt iteration and one conjugate gradient iteration.

Table 7. Time taken by CPU and GPU implementations. (CP denotes Camera Parameters).

Dataset No.	*Without-Schur* Complement				*With-Schur* Complement			
	CPU (secs)		GPU (secs)		CPU (secs)		GPU (secs)	
	8 CP	9 CP	8 CP	9 CP	8 CP	9 CP	8 CP	9 CP
1	0.016	0.018	0.004	0.004	0.016	0.019	0.004	0.005
2	0.069	0.080	0.009	0.010	0.074	0.083	0.011	0.009
3	0.176	0.193	0.015	0.018	0.192	0.214	0.019	0.022
4	0.288	0.325	0.023	0.028	0.217	0.354	0.020	0.032
5	0.392	0.439	0.043	0.048	0.415	0.490	0.049	0.054
6	0.426	0.476	0.046	0.052	0.454	0.507	0.052	0.059
7	0.510	0.567	0.055	0.062	0.538	0.602	0.063	0.070
8	0.691	0.787	0.056	0.065	0.748	0.835	0.062	0.073
9	1.127	1.281	0.099	0.114	1.203	1.347	0.112	0.130
10	2.188	2.430	0.168	0.198	2.299	2.687	0.183	0.219

camera parameter. However, the rate of increase in time is higher in the CPU implementations compared to the GPU implementations. One of the reasons is that in GPU implementations, more of the computations are concurrently executed, resulting in higher throughput than in CPU implementations.

6.2.2. PBA CUDA and Proposed CUDA Implementation Comparison

In addition to the above speedup analysis, we discuss the preliminary speedup comparison between PBA CUDA implementation and proposed CUDA implementation. The PBA CUDA and proposed CUDA implementations converge to a different mean square error with a different number of LM iterations and conjugate gradient iterations. This is because the proposed CUDA implementation has different configurations, like using explicit Jacobian and nine camera parameters. As a result, the total LM iterations in the proposed CUDA implementation are modified to attain similar or better mean square error as the PBA CUDA implementation.

The analysis shows that the speedup varies between the proposed CUDA implementation and the PBA CUDA implementation to converge to a specific mean square error, with the proposed CUDA implementation being performant compared to the PBA CUDA implementation in most cases. The total number of conjugate gradient steps required by the PBA CUDA implementation is higher than the proposed CUDA implementation. This indicates that the time taken by the PBA CUDA implementation per iteration of the LM step, or the conjugate gradient iteration is less than the proposed CUDA implementation. Various advanced optimization techniques, like improving the memory throughput and minimizing the warp divergence, can be implemented in the proposed CUDA implementation. As a result, further optimization of the proposed CUDA code will improve the speedup compared to the PBA CUDA implementation.

7. Conclusion and Future Work

The camera parameters and 3D points play a vital role in scene reconstruction. Refining of the intrinsic and extrinsic parameters of the camera and the 3D points through bundle adjustment algorithm provides accurate information about the depth, orientation, and alignment of the objects in the scene. Refinement of the camera parameters and 3D points involves solving of the LM algorithm which is computationally expensive and proportional to the number of images, points, and projections. In this paper, the accuracy of the bundle adjustment algorithm is improved by minimizing the reprojection error. In addition, the computation cost of the algorithm is addressed by parallel programming using CUDA programming model. Overall, the accuracy of the refinement is studied for obtaining improved scene reconstruction and the computation cost of the algorithm is improved as it is computationally expensive for refining larger datasets.

An additional radial distortion parameter and explicit Jacobian computation are analyzed and demonstrated to have better convergence. The BA algorithm is

also implemented on GPUs using the CUDA programming model and demonstrated to have a significant speedup compared to the sequential implementation. In addition, we also demonstrate the use of various CUDA features that improve the performance of the BA algorithm by reducing the computational time. The concurrency of the computations in the BA algorithm is exploited using CUDA streams, resulting in asynchronous execution. We have used atomic operations to read and write data from/to the same memory location by different threads without race conditions.

The use of complex preconditioners that would improve the convergence in the conjugate gradient method can be studied as part of future work. The increased computational time taken by the complex preconditioners can be addressed using GPUs. As aforementioned, the GPU implementation in the paper involves basic optimization techniques to improve the speedup of the GPU code. Further advanced optimization techniques involving data layout and utilizing shared and register memory effectively can be implemented in the CUDA code. In addition, comparing the proposed CUDA code with the state-of-the-art GPU implementations can provide further analysis of the algorithm and its behavior on the GPUs.

Acknowledgements

We want to thank Sameer Agarwal *et al.* [7] for releasing the datasets and additional information through the online URL https://grail.cs.washington.edu/projects/bal/ and Changchang Wu *et al.* [12] for releasing software and additional information through the online URL https://grail.cs.washington.edu/projects/mcba/.

We would also like to thank NVIDIA corporation for providing access to the Selene cluster for development purposes.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Chen, C.I., Sargent, D., Tsai, C.M., Wang, Y.F. and Koppel, D. (2009) Uniscale Multi-View Registration Using Double Dog-Leg Method. *Medical Imaging* 2009: *Visualization*, *Image-Guided Procedures*, *and Modeling*, Vol. 7261, 468-477. https://doi.org/10.1117/12.810966

[2] Favalli, M., Fornaciai, A., Isola, I., Tarquini, S. and Nannipieri, L. (2012) Multiview 3D Reconstruction in Geosciences. *Computers & Geosciences*, **44**, 168-176. https://doi.org/10.1016/j.cageo.2011.09.012

[3] Lowe, D.G. (2004) Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, **60**, 91-110. https://doi.org/10.1023/B:VISI.0000029664.99615.94

[4] Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R. and Wu, A.Y. (1998) An Op-

timal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. *Journal of the ACM* (*JACM*), **45**, 891-923. https://doi.org/10.1145/293347.293348

[5] Agarwal, S., Snavely, N., Simon, I., Seitz, S.M. and Szeliski, R. (2009) Building Rome in a Day. 2009 *IEEE* 12*th International Conference on Computer Vision*, Kyoto, 29 September-2 October 2009, 72-79. https://doi.org/10.1109/ICCV.2009.5459148

[6] Lourakis, M.I. and Argyros, A.A. (2009) SBA: A Software Package for Generic Sparse Bundle Adjustment. *ACM Transactions on Mathematical Software* (*TOMS*), **36**, 1-30. https://doi.org/10.1145/1486525.1486527

[7] Agarwal, S., Snavely, N., Seitz, S.M. and Szeliski, R. (2010) Bundle Adjustment in the Large. *Computer Vision ECCV* 2010: 11*th European Conference on Computer Vision*, Heraklion, 5-11 September 2010, 29-42. https://doi.org/10.1007/978-3-642-15552-9_3

[8] Björck, Å. (1996) Numerical Methods for Least Squares Problems. Society for Industrial and Applied Mathematics, Philadelphia.

[9] Curry, H.B. (1944) The Method of Steepest Descent for Non-Linear Minimization Problems. *Quarterly of Applied Mathematics*, **2**, 258-261. https://doi.org/10.1090/qam/10667

[10] Levenberg, K. (1944) A Method for the Solution of Certain Non-Linear Problems in Least Squares. *Quarterly of Applied Mathematics*, **2**, 164-168. https://doi.org/10.1090/qam/10666

[11] Marquardt, D.W. (1963) An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *Journal of the Society for Industrial and Applied Mathematics*, **11**, 431-441. https://doi.org/10.1137/0111030

[12] Wu, C., Agarwal, S., Curless, B. and Seitz, S.M. (2011) Multicore Bundle Adjustment. *CVPR* 2011, Colorado Springs, 20-25 June 2011, 3057-3064. https://doi.org/10.1109/CVPR.2011.5995552

[13] Zheng, M., Zhou, S., Xiong, X. and Zhu, J. (2017) A New GPU Bundle Adjustment Method for Large-Scale Data. *Photogrammetric Engineering & Remote Sensing*, **83**, 633-641. https://doi.org/10.14358/PERS.83.9.633

[14] On Derivatives—Ceres Solver (n.d.). http://ceres-solver.org/derivatives.html

[15] Snavely, N., Seitz, S.M. and Szeliski, R. (2006) Photo Tourism: Exploring Photo Collections in 3D. *ACM SIGGRAPH* 2006 *Papers*, Boston, 30 July-3 August 2006, 835-846.

[16] Snavely, N., Seitz, S.M. andSzeliski, R. (2008) Skeletal Graphs for Efficient Structure from Motion. 2008 *IEEE Conference on Computer Vision and Pattern Recognition*, Anchorage, 23-28 June 2008, 1-8. https://doi.org/10.1109/CVPR.2008.4587678

[17] Byröd, M. and Åström, K. (2010) Conjugate Gradient Bundle Adjustment. *Computer Vision ECCV* 2010: 11*th European Conference on Computer Vision*, Heraklion, 5-11 September 2010, 114-127. https://doi.org/10.1007/978-3-642-15552-9_9

[18] Byröd, M. and Åström, K. (2009) Bundle Adjustment Using Conjugate Gradients with Multiscale Preconditioning. *British Machine Vision Conference*, *BMVC* 2009, London, 7-10 September 2009, 1-10. https://doi.org/10.5244/C.23.36

[19] Jeong, Y., Nister, D., Steedly, D., Szeliski, R. and Kweon, I.S. (2011) Pushing the Envelope of Modern Methods for Bundle Adjustment. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **34**, 1605-1617. https://doi.org/10.1109/TPAMI.2011.256

[20] Sameer, A. and Mierle, K. (2012) Ceres Solver.

[21] Choudhary, S., Gupta, S. and Narayanan, P.J. (2012) Practical Time Bundle Ad-

justment for 3D Reconstruction on the GPU. *Trends and Topics in Computer Vision*: *ECCV* 2010 *Workshops*, Heraklion, 10-11 September 2010, 423-435. https://doi.org/10.1007/978-3-642-35740-4_33

[22] Tomov, S., Dongarra, J., Volkov, V. and Demmel, J. (2009) Magma Library. Univ. of Tennessee and Univ. of California, Knoxville.

[23] Nielsen, H.B. (1999) Damping Parameter in Marquardt's Method (Vol. 248) IMM.

[24] Dao Khanh, H. and Dinchev, L. (2020, November 8) Bundle Adjustment—Part 1: Jacobians. Telesens. https://telesens.co/2016/10/13/bundle-adjustment-part-1-jacobians/

[25] https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula

[26] MathWorks, I. (2022, January 9) Symbolic Math Toolbox. https://www.mathworks.com/help/symbolic/

[27] Nvidia, C. (2022) Programming Guide: CUDA Toolkit Documentation.

[28] Nvidia, C.U.D.A. (2008) Cublas Library. NVIDIA Corporation, Santa Clara.

[29] NVIDIA Nsight Systems (2020, May) NVIDIA Documentation Center. https://docs.nvidia.com/nsight-systems/2020.3/profiling/index.html