Journal of Software Engineering and Applications, 2024, 17, 172-201
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

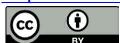
Optimized CUDA Implementation to Improve the Performance of Bundle Adjustment Algorithm on GPUs

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, mcinroy@uwyo.edu

How to cite this paper: Kommera, P.R., Muknahallipatna, S.S. and McInroy, J.E. (2024) Optimized CUDA Implementation to Improve the Performance of Bundle Adjustment Algorithm on GPUs. *Journal of Software Engineering and Applications*, **17**, 172-201.
https://doi.org/10.4236/jsea.2024.174010

Received: March 28, 2024
Accepted: April 25, 2024
Published: April 28, 2024

Copyright © 2024 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

The 3D reconstruction pipeline uses the Bundle Adjustment algorithm to refine the camera and point parameters. The Bundle Adjustment algorithm is a compute-intensive algorithm, and many researchers have improved its performance by implementing the algorithm on GPUs. In the previous research work, "Improving Accuracy and Computational Burden of Bundle Adjustment Algorithm using GPUs," the authors demonstrated first the Bundle Adjustment algorithmic performance improvement by reducing the mean square error using an additional radial distorting parameter and explicitly computed analytical derivatives and reducing the computational burden of the Bundle Adjustment algorithm using GPUs. The naïve implementation of the CUDA code, a speedup of 10× for the largest dataset of 13,678 cameras, 4,455,747 points, and 28,975,571 projections was achieved. In this paper, we present the optimization of the Bundle Adjustment algorithm CUDA code on GPUs to achieve higher speedup. We propose a new data memory layout for the parameters in the Bundle Adjustment algorithm, resulting in contiguous memory access. We demonstrate that it improves the memory throughput on the GPUs, thereby improving the overall performance. We also demonstrate an increase in the computational throughput of the algorithm by optimizing the CUDA kernels to utilize the GPU resources effectively. A comparative performance study of explicitly computing an algorithm parameter versus using the Jacobians instead is presented. In the previous work, the Bundle Adjustment algorithm failed to converge for certain datasets due to several block matrices of the cameras in the augmented normal equation, resulting in rank-deficient matrices. In this work, we identify the cameras that cause rank-deficient matrices and preprocess the datasets to ensure the convergence of the BA algorithm. Our optimized CUDA implementation achieves convergence of the Bundle Adjustment algorithm in around 22 seconds for the

largest dataset compared to 654 seconds for the sequential implementation, resulting in a speedup of 30×. Our optimized CUDA implementation presented in this paper has achieved a 3× speedup for the largest dataset compared to the previous naïve CUDA implementation.

Keywords

Scene Reconstruction, Bundle Adjustment, Levenberg-Marquardt, Non-Linear Least Squares, Memory Throughput, Computational Throughput, Contiguous Memory Access, CUDA Optimization

1. Introduction

In the previous article [1], the authors presented the use of the second radial distorting coefficient and explicitly computed analytical derivatives to modify the Adjustment (BA) algorithm to reduce the mean square error. Using the second radial distorting parameter to reduce the mean square error resulted in additional computations. A naïve CUDA implementation on graphics processing units (GPUs) addressed the increased computational burden. The computational performance of the modified BA algorithm [1] for the largest dataset with the naïve CUDA implementation resulted in a speedup of 10×. The naïve CUDA implementation performance of the modified BA algorithm was insufficient for large datasets due to not addressing the computational and memory throughput issues associated with execution on GPUs.

The performance of the GPU execution greatly depends on the computational and memory throughput. The computational throughput depends on the level of concurrency that can be achieved in computing different mathematical operations asynchronously. The memory throughput depends on the rate at which the data can be accessed from the memory and the total number of memory transactions. The time taken to access data from memory is around two orders of magnitude compared to the time taken to perform a mathematical computation, which is less than one order of magnitude. This paper proposes techniques to improve the computational and memory throughput of the modified BA algorithm [1].

We demonstrate an optimization technique that employs shared memory on GPUs to compute intermediate mathematical operations asynchronously before obtaining the final output from the intermediate results. This optimization technique, which improves computational throughput, is widely used to distribute the workload effectively across threads in the blocks on GPUs, thereby improving GPU utilization.

We also propose a new data layout for all the mathematical variables represented as block matrices and vectors. The proposed data layout will have elements of each vector/matrix of each mathematical variable distributed across the memory. Accessing the data from the proposed memory layout will signif-

icantly reduce the overall memory transactions, thereby improving the memory throughput.

In addition to the proposed new data layout and CUDA kernel optimization, we also analyze the datasets provided in [2] to address the non-convergence issue mentioned in [1]. Thirty-two datasets in the large datasets [2] do not converge from their initial mean square error. This paper investigates the reasons for the non-convergence and proposes modifications to the datasets so that the modified BA algorithm can converge. In addition, we extend our performance studies to datasets with 10,000 images and millions of points and projections.

The rest of the paper is organized as follows. Section 2 talks about various implementations of the BA algorithm on GPUs. Section 3 briefly introduces the BA algorithm, its mathematical representation, and the modified BA algorithm from [1]. Section 4 provides an introduction to the GPU hardware and performance. Section 5 details the CUDA optimization and the proposed data layout. Section 6 provides information about the datasets and the proposed modifications to ensure all the datasets converge from their initial mean square error. Section 7 demonstrates the results and provides information about the performance of the implementation compared to the sequential and GPU versions. Finally, the paper summarizes the findings and puts forward the scope of future work in the conclusion and future work section.

2. Literature Review

Implementation in [3] developed a GPU version of the BA algorithm that employs the exact-step Levenberg-Marquardt (LM) method [4] using Compressed Column Storage (CCS) format. The computationally intensive left-hand side of the augmented normal equations is calculated on the GPU. In contrast, the equations' relatively less computationally intensive right-hand side is computed on the central processing units (CPU). The computationally intensive linear systems are solved using the MAGMA library [5]. The implementation has stored all the mathematical parameters on the GPUs, resulting in higher memory requirements. In addition, Ceres Solver [6], a C++ library to solve the non-linear least squares problem for the exact-step LM method, is used. The Ceres Solver library only has CUDA support for the dense Cholesky decomposition variant and the Schur complement. The library does not support CUDA sparse computations in the exact-step LM method; instead uses third-party libraries like Apple's Accelerate framework [7] and Eigen's sparse linear solvers [8]. The exact-step method is found to be optimal for smaller datasets but was found to be computationally expensive for larger datasets due to the use of Cholesky factorization.

A GPU version of the inexact-step LM method [2] [9] [10] using a block Jacobi preconditioner is developed in Parallel Bundle Adjustment (PBA) [11]. 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 sparse matrices in the PBA implementation are stored in Block Compressed Sparse Row (BCSR) format. The implementation extensively uses texture object functions of the CUDA runtime application programming interface (API). The PBA implementation has shown a significant performance boost compared to single-core executions.

Ceres solver [6] library also computes the inexact-step LM method with the CUDA-enabled preconditioner conjugate gradient (PCG) method. The library supports GPU implementation of a few preconditioners, like Jacobi and Schur Jacobi. The library can refine around six to nine camera parameters. The library does not explicitly compute the augmented normal equation; instead, it uses the Jacobians stored in Compressed Row Sparse (CSR) format.

Implementation in [12] has refined 11 camera parameters, consisting of three rotational elements, three translational elements, one focal length, two corrections of the principal point, and two radial distortions using the inexact-step LM method. The time-consuming portions of the algorithm are simulated on the GPU, and the performance is evaluated on datasets with fewer points and projections, demonstrating performance improvement compared to the PBA implementation.

Unlike in PBA implementation, our previous work has implemented the BA algorithm on CPUs and improved the accuracy of the minimization by using additional radial distortion and explicit Jacobian computation. The PBA implementation [11] exploited the block structure in the BA algorithm with texture memory fetching and shared memory. The texture reference management functions are deprecated in the latest CUDA runtime APIs [13]. As a result, the PBA implementation cannot be executed using the texture memory. In this paper, we adopt a similar strategy from the PBA implementation for using shared memory and extend our previous work [1] by further optimizing the CUDA kernels.

In addition, we also extend our previous work [1] by improving the memory throughput. Our previous implementation and implementations in PBA and Ceres Solver all store the mathematical parameters in block structures. Elements of each mathematical variable represented in the form of block vectors/matrices are stored per the spatial locality. In this paper, we demonstrate the drawback of storing the matrices/vectors of each block element in continuous memory, resulting in lower memory throughput. We propose a new data memory layout to reduce the total memory transactions and improve the algorithm's performance. In addition, we also illustrate the impact of the proposed memory layout on the algorithm's overall performance by a comparative study of the optimized CUDA with the naïve CUDA implementation in [1].

3. Brief Discussion of the Modified Bundle Adjustment Algorithm

Minimizing the reprojection error involves solving the augmented normal equation, as shown in Equation (1), which can be represented as a linear system of

equations as in Equation (2).

$$\left(\boldsymbol{J}^{\mathrm{T}}\boldsymbol{J}+\mu\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}\right)\boldsymbol{\delta}=-\boldsymbol{J}^{\mathrm{T}}\boldsymbol{e} \tag{1}$$

$$\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b} \tag{2}$$

where,

$\boldsymbol{J}=\partial f\left(\boldsymbol{P}\right)/\partial\boldsymbol{P}$ is a Jacobian of the projection function $f\left(\boldsymbol{P}\right)$ [1];

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

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

$\boldsymbol{D}$ is a non-negative diagonal matrix formulated as the square root of the diagonal of the matrix $\boldsymbol{J}^{\mathrm{T}}\boldsymbol{J}$ [2].

μ is the positive damping parameter used to control the regularization;

$\boldsymbol{A}=\boldsymbol{J}^{\mathrm{T}}\boldsymbol{J}+\mu\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}$ 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.

The augmented normal equation can be represented [1] [4] of the camera and points sections in the matrix notation as in Equation (3).

$$\begin{bmatrix}\boldsymbol{U}_{\mu} & \boldsymbol{W}\\ \boldsymbol{W}^{\mathrm{T}} & \boldsymbol{V}_{\mu}\end{bmatrix}\begin{bmatrix}\boldsymbol{\delta}_{c}\\ \boldsymbol{\delta}_{p}\end{bmatrix}=\begin{bmatrix}\boldsymbol{\epsilon}_{c}\\ \boldsymbol{\epsilon}_{p}\end{bmatrix} \tag{3}$$

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

where,

c represents the camera section;

p represents the point section.

$\boldsymbol{U}_{\mu}$ and $\boldsymbol{V}_{\mu}$ are block diagonal matrices; $\boldsymbol{W}$ is a block sparse matrix; $\boldsymbol{\delta}_{c}$, $\boldsymbol{\delta}_{p}$, $\boldsymbol{\epsilon}_{c}$, and $\boldsymbol{\epsilon}_{p}$ are the block vectors.

The preconditioned conjugate gradient (PCG) method is an iterative solver used to solve the system of linear equations. The PCG method employs a preconditioner for better and faster convergence and involves a significantly higher number of vector-vector and matrix-vector computations per iteration [12]. The PCG method is employed on the augmented normal equation represented by Equation (3). The PCG method can also be applied to a matrix with reduced dimensions by multiplying both sides of Equation (3) with a block matrix, as illustrated in [4], which results in Equation (4) and Equation (5).

$$\left(\boldsymbol{U}-\boldsymbol{W}\boldsymbol{V}_{\mu}^{-1}\boldsymbol{W}^{\mathrm{T}}\right)\boldsymbol{\delta}_{c}=\boldsymbol{\epsilon}_{c}-\boldsymbol{W}\boldsymbol{V}_{\mu}^{-1}\boldsymbol{\epsilon}_{p} \tag{4}$$

$$\boldsymbol{V}_{\mu}\boldsymbol{\delta}_{p}=\boldsymbol{\epsilon}_{p}-\boldsymbol{W}^{\mathrm{T}}\boldsymbol{\delta}_{c} \tag{5}$$

In this research, similar to the representation in [1], solving Equation (3) directly for the solution vector is called *Without-Schur* complement, and solving for the solution vector using the Schur complement representation in Equation (4) and Equation (5) is called *With-Schur* complement.

The modified BA algorithm proposed in [1] uses an additional radial distortion coefficient as part of the camera parameters. An additional camera parame-

ter increases the dimension of the camera section of the Jacobian as represented in Equation (6) (additional radial distortion coefficient parameter is represented in bold).

$$\frac{\partial f\left(\boldsymbol{P}_{ij}\right)}{\partial \boldsymbol{c}^{j}}=\begin{bmatrix}\frac{\partial x_{ij}}{\partial c^{j}}\\ \frac{\partial y_{ij}}{\partial c^{j}}\end{bmatrix}=\begin{bmatrix}\frac{\partial x_{ij}}{\partial k_1^j} & \frac{\partial x_{ij}}{\partial k_2^j} & \frac{\partial x_{ij}}{\partial k_3^j} & \frac{\partial x_{ij}}{\partial t_1^j} & \frac{\partial x_{ij}}{\partial t_2^j} & \frac{\partial x_{ij}}{\partial t_3^j} & \frac{\partial x_{ij}}{\partial f^j} & \frac{\partial x_{ij}}{\partial K_1{}^j} & \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{K}_2^j}\\ \frac{\partial y_{ij}}{\partial k_1^j} & \frac{\partial y_{ij}}{\partial k_2^j} & \frac{\partial y_{ij}}{\partial k_3^j} & \frac{\partial y_{ij}}{\partial t_1^j} & \frac{\partial y_{ij}}{\partial t_2^j} & \frac{\partial y_{ij}}{\partial t_3^j} & \frac{\partial y_{ij}}{\partial f^j} & \frac{\partial y_{ij}}{\partial K_1{}^j} & \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{K}_2^j}\end{bmatrix} \quad (6)$$

where,

$\left(x_{ij}, y_{ij}\right)$ is the projection in the 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*

The increase in the dimension of the camera section of the Jacobian, in turn, increases the size of all the parameters in the augmented normal equation, as represented in [1]. In addition, the modified BA algorithm uses 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 without approximations using the "diff" command in Matlab Symbolic Math Toolbox [14] and hardcoding the projection function into the code.

4. GPU Hardware and Performance

Central processing units (CPUs) and graphics processing units (GPUs) are two different types of processors widely used for computational purposes. The hardware configurations of CPUs and GPUs differ significantly, making them ideal for specific problems. CPUs have fewer cores with higher clock rates, whereas GPUs have a larger number of cores with slower clock rates than CPUs. In addition, the CPUs' cores contain complex pipelines, making them ideal for decision-making instructions like conditional loops. Whereas the cores in GPUs are lightweight and ideal for basic mathematical operations. A large number of simple cores and their ability to execute instructions concurrently make the GPUs ideal for large independent mathematical operations. A fewer number of complex cores with higher clock rates make CPUs ideal for branching and conditional loops and input/output operations.

As mentioned, the concurrency offered by the GPUs makes them ideal for large-scale independent mathematical operations. In addition, the fundamental properties of the GPUs offering concurrency and memory throughput are the same across different GPU architectures. The differences across the GPU architectures are mainly in the level of concurrency, memory throughput, and a few

additional features. The number of computing cores in the A30 GPU is 3584, whereas the total number of cores in the latest NVIDIA GPU H100 is 7296, resulting in approximately twice the level of concurrency compared [15] to an A30 GPU. The difference across the GPUs is also in the memory bandwidth. The A30 GPU [15] has a memory bandwidth of 933.1 GB/s, whereas the H100 GPU has around 1280 GB/s. This difference in the memory bandwidth results in higher memory throughput in the H100 GPU compared to the A30 GPU. In addition, there are several other differences, such as the total streaming multiprocessors (SMs), memory bus width, increased size of different memory categories, and boost clock speed, that are improved in the latest GPU architectures. The increased size of different memory on the GPUs would result in increased caching and improved bandwidth, but the core functionality is the same across different GPU architectures. The latest GPUs also include new features [16], like tensor cores and multi-instance GPUs (MIG), which are beyond the scope of the paper and do not impact the proposed methodologies in this paper. Nonetheless, the total number of cores and the memory throughput play a vital role in performance improvement across different GPU architectures. Irrespective of the difference in the GPU architectures, the underlying principles of the computational throughput and the memory throughput across the GPU architectures are the same. As a result, the same code base that was executed on an older GPU can be readily executed on the latest GPUs without any changes to the code and achieve a significant performance boost based on the hardware improvements alone.

GPUs are equipped with thousands of cores that can execute operations concurrently. On the other hand, the number of independent computations available in the algorithm allows to take advantage of the concurrency on the GPUs. As mentioned in the implementation [1], the independent nature of the camera and point sections in the algorithm enables the execution of each mathematical operation concurrently, thereby reducing the total time taken for all computations. In addition, the block-based structures across both the camera and point sections of each matrix and vector enable further concurrency in the matrix-vector multiplications.

In addition to the computational concurrency, memory throughput also plays a vital role in achieving better performance on the GPU. The memory throughput depends on the access time and the total number of memory transactions required for the computations. Effective utilization of the caching techniques on the GPU using a register shared, and global memory will reduce the total time taken to access the data. In addition to the caching techniques, reducing the total number of memory transactions required for the computations also reduces the total computational time. A detailed understanding of the memory access patterns is required to utilize the required memory transactions effectively.

4.1. Memory Access Patterns

The computational and memory operations are issued per warp in the CUDA programming model. A warp is a group of 32 threads that concurrently issues

and executes the same instruction. While executing the memory instructions, each thread in a warp issues a request to access a memory address. All the requests from 32 threads in the warp are grouped and processed as a single memory transaction. Whenever a memory transaction is requested, based on the spatial locality, the first memory address is accessed and loads contiguous data from that address. The amount of contiguous memory segments transferred in a single memory transaction depends on the granularity of the cache line which is mostly 128-bytes or 32-bytes. In this research work, we assume the granularity to be 128 bytes, and each thread in a warp is requesting 4-byte data, and consider two basic access patterns that could arise in any memory access.

4.1.1. Coalesced Memory Access

In the coalesced memory access pattern, each thread in a warp accesses contiguous memory. As a result, all 32 threads access 128 bytes, which is contiguous in memory, as in Figure 1.

As the granularity is assumed to be 128 bytes, the entire 128 bytes requested by the warp are addressed in a single memory transaction. This is an ideal case where one memory request fulfills the data requests by all 32 threads in a warp.

4.1.2. Strided Memory Access

In the strided memory access pattern, each thread in a warp accesses data from a different locality in the memory. As a result, multiple memory transactions are required to fetch all the data requested by the 32 threads in a warp. Figure 2 provides an example of a strided access pattern in which the first 16 threads access data from one memory segment, and the next 16 threads access data from a different memory segment.

In this example of strided access, the first 16 threads access 64 bytes of contiguous memory from a 128-byte memory segment in the memory, and the next 16 threads access the remaining 64 bytes of contiguous memory from the next 128-byte memory segment. As a result, two memory transactions are required to fetch the data requested by all the 32 threads in a warp.

There are many different strided access patterns possible. In the worst-case scenario, each thread in the warp will access data from different 128-byte memory segments, requiring 32 memory transactions to access the data requested by all 32 threads in a warp.

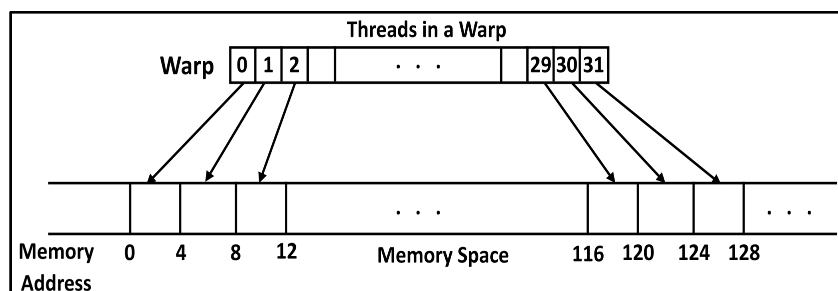


Figure 1. Pictorial representation of the coalesced memory access pattern.

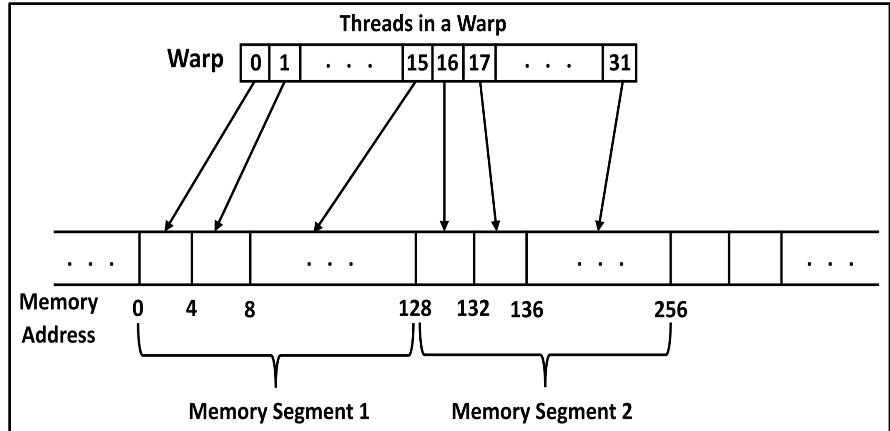


Figure 2. Pictorial representation of the strided memory access pattern accessing two memory segments.

Overall, the ideal case results in one memory transaction, whereas the strided access results in 2 - 32 memory transactions based on the degree of data distribution in the memory. In this paper, we propose a new data layout to improve the access pattern and reduce the total memory transactions.

5. Proposed Implementations

In the BA algorithm, all the elements of each matrix and vector are represented as blocks. For example, the block-matrix $\boldsymbol{U}_\mu$ and $\boldsymbol{V}_\mu$ are represented as in Equation (7).

$$\boldsymbol{U}_\mu = \begin{bmatrix} \boldsymbol{U}_1 & 0 & 0 & 0 \\ 0 & \boldsymbol{U}_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \boldsymbol{U}_m \end{bmatrix}; \boldsymbol{V}_\mu = \begin{bmatrix} \boldsymbol{V}_1 & 0 & 0 & 0 \\ 0 & \boldsymbol{V}_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \boldsymbol{V}_m \end{bmatrix} \tag{7}$$

where,

$\boldsymbol{U}_i, i = 1, 2, \cdots, m$ are the blocks of m cameras each of size 9 × 9;

$\boldsymbol{V}_j, j = 1, 2, \cdots, n$ are the blocks of n points each of size 3 × 3.

Similarly, each vector element is stored in the form of blocks, as shown in Equation (8).

$$\boldsymbol{\epsilon}_c = \begin{bmatrix} \boldsymbol{\epsilon}_c^1 \\ \boldsymbol{\epsilon}_c^2 \\ \vdots \\ \boldsymbol{\epsilon}_c^m \end{bmatrix}; \boldsymbol{\epsilon}_p = \begin{bmatrix} \boldsymbol{\epsilon}_p^1 \\ \boldsymbol{\epsilon}_p^2 \\ \vdots \\ \boldsymbol{\epsilon}_p^n \end{bmatrix} \tag{8}$$

where,

$\boldsymbol{\epsilon}_c^i, i = 1, 2, \cdots, m$ are the blocks of m cameras each of size 9 × 1;

$\boldsymbol{\epsilon}_p^i, i = 1, 2, \cdots, n$ are the blocks of n points each of size 3 × 1.

In the BA algorithm, formulation of the augmented normal equation as in Equation (3) results in significant matrix-matrix and matrix-vector computations. In addition, the PCG method involves matrix-vector and vector-vector

computations. The highly optimized cuBLAS executes vector-vector operations [17] library, whereas the sparse nature of the matrix-matrix and matrix-vector operations restrict the use of the cuBLAS library. As a result, CUDA kernels are developed to compute the matrix-matrix and matrix-vector operations. In our previous work [1], we implemented basic CUDA optimization techniques to improve the BA performance on the GPUs. In this paper, we demonstrate additional optimization techniques to improve the performance of the CUDA kernels involving matrix-vector multiplications.

5.1. CUDA Kernel Optimization

In the previous implementation [1], we observed that the use of atomic operations introduced race conditions in higher datasets due to the sensitivity of the data precision. To address the sensitivity of the data precision, we propose eliminating the use of atomic operations. However, eliminating atomic operations will not allow navigation through each projection, obtain the camera and point information of that projection, and compute all the parameters of a specific camera and point configuration as implemented in [1]. Instead, to compute a particular parameter belonging to a specific camera, we gather all the points and projections information about that camera and compute it on a single block. Similarly, we gather all the camera and projection information about a point to compute a particular parameter. This change does not cause any race conditions and produces accurate results.

Naïve CUDA implementation [1] describes the optimal thread and block configurations that can be employed based on the computation of the camera, point, and projection sections. Unlike in [1], where the computation of each camera is distributed across each thread, we perform the computation of each camera on a block with threads computing intermediate results. As the number of cameras is fewer and the number of points is significantly higher, assigning computations of each camera to a block is found to be optimal. In this configuration, each thread of a block computes a temporary solution of an operation. The solutions from all the threads are aggregated in the end to compute a parameter for a camera.

For example, let us compute the first camera block-matrix $\boldsymbol{U}_1$ in the augmented normal equation, which can be represented [1] as in Equation (9).

$$\boldsymbol{U}_1 = \sum_{i=1}^{n} \boldsymbol{J}_{i1}^{\mathrm{T}} \boldsymbol{J}_{i1} \tag{9}$$

where,

$\boldsymbol{J}_{i1}, i = 1, 2, \cdots, n$ are the block-Jacobian matrices of n points each of size 2 × 9.

Computing the $\boldsymbol{U}_1$ block matrix of the camera section involves iterating through n points. As a result, all the points are divided among the threads in a block, and each thread computes the summation of the $\boldsymbol{J}_{i1}^{\mathrm{T}} \boldsymbol{J}_{i1}$ for those set of points assigned to it. And in the end, all the intermediate solutions are aggregated on a single thread. **Figure 3** provides a pictorial representation of computing $\boldsymbol{U}_1$ with 9 points on a block that has 3 threads.

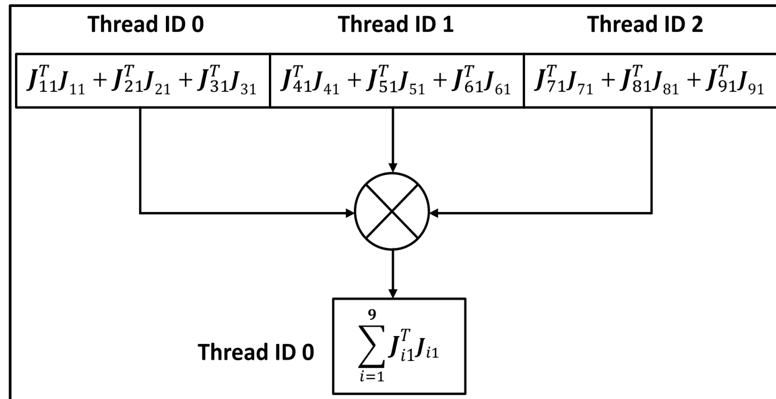


Figure 3. Pictorial representation of computing U_1 with 9 points on a block and 3 threads.

The above optimization is ideal for computing the parameters of the camera section, as it iterates through a larger number of points. The same methodology is found to be non-optimal for computing parameters of the point section as we have a higher number of points and fewer cameras to iterate through. A higher number of points invokes more blocks, which in turn serializes the execution. As a result, all the parameters of the points section are computed directly on each thread across the blocks. In addition, unrolling the points on each thread is employed to compute the parameters of multiple points on a single thread, thereby reducing the total number of blocks.

The CUDA kernel optimization studied in this paper aims at improving the concurrency of computing the camera parameters by distributing the computations across threads in a block and by effective utilization of shared memory. These properties of performance improvement using concurrency and the use of shared memory are the same across different GPU architectures. The difference only lies in the level of concurrency and the size of shared memory. As a result, the CUDA kernel optimization studied in this paper is applicable across different GPU architectures without any changes to the code implementation.

In the current implementations [1], each block of a matrix is stored in a contiguous memory, as represented in **Figure 4**. In turn, all the elements of each block are contiguous in memory.

In the above figure, the dimension of each block-Jacobian matrix is 2×9. This layout results in more memory transactions due to the strided access pattern. In this paper, we analyze the current memory layout and propose a new data layout that would reduce the degree of strided access and reduce the total memory transactions.

5.2. Proposed Memory Layout

The data layout, as mentioned above, results in strided access as each element of respective blocks is in memory addresses far from each other. For example, consider a multiplication operation [1] between a block matrix and a block vector, as in Equation (10).

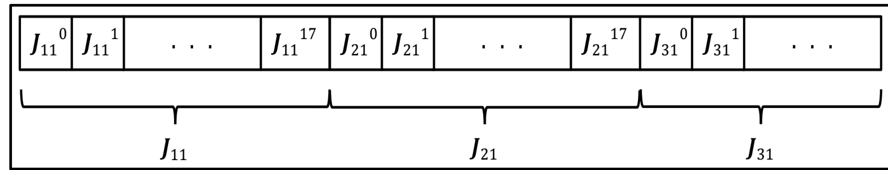


Figure 4. Layout of Jacobian-matrices $J_{i1}, i = 1, \cdots, n$ in memory.

$$U_\mu p_c^k = \begin{bmatrix} U_1 & 0 & 0 & 0 \\ 0 & U_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & U_m \end{bmatrix} \begin{bmatrix} p_c^1 \\ p_c^2 \\ \vdots \\ p_c^m \end{bmatrix} \tag{10}$$

where,

$U_i, i = 1, 2, \cdots, m$ are the block-matrices each of size 9 × 9;

$p_c^i, i = 1, 2, \cdots, m$ are the block-vectors each of size 9 × 1.

In the current implementation, all the elements of each block of matrix and vector are stored in contiguous memory, as in **Figure 5**.

In the CUDA kernel, each block multiplication can be assigned to a single thread. As a result, thread 0 will access U_1, thread 1 will access U_2 and so on. Analyzing further, when thread 0 accesses the first element in U_1, thread 1 will access the first element in U_2 which is 9 × 9 memory spaces away from the first element of U_1. As all the elements of each block-matrix U_i are stored in contiguous memory, each thread will end up having strided access to fetch the element from memory, as shown in **Figure 6**. This results in multiple memory transactions per request from a warp.

In this paper, we propose to modify the data placement in memory so that adjacent threads can access contiguous memory, thereby reducing the strided access and memory transactions required per request. To access contiguous memory, we propose the *continuous-element* data layout where each element from a particular index across the blocks is stored in contiguous memory, as depicted in **Figure 7** and **Figure 8**.

Using the proposed *continuous-element* data layout, computing the matrix-vector multiplication in Equation (10) will result in a contiguous memory access pattern, as shown in **Figure 9**. We propose to change the data layout to the *continuous-element* format for all the matrix and vector variables in the BA algorithm. The use of the proposed *continuous-element* format would reduce the total memory transactions per request and improve the GPU performance.

The proposed *continuous-element* data layout aims at improving the memory throughput which employs similar fundamental principles across different GPU architectures. As mentioned earlier, the difference lies in the magnitude of the memory access, *i.e.*, the memory bandwidth, which describes the time required to access the memory. Whereas the functional properties describing the total number of memory transactions per request are the same across different GPU architectures. With this proposed data layout, we are reducing the total number

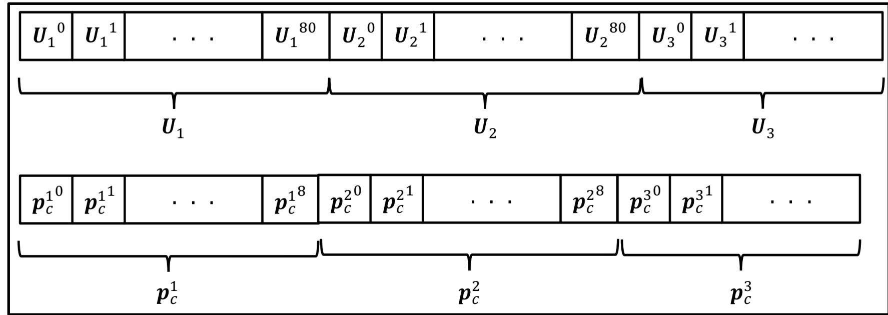


Figure 5. Memory layout of block-matrix U_i and block-vector p_c^i.

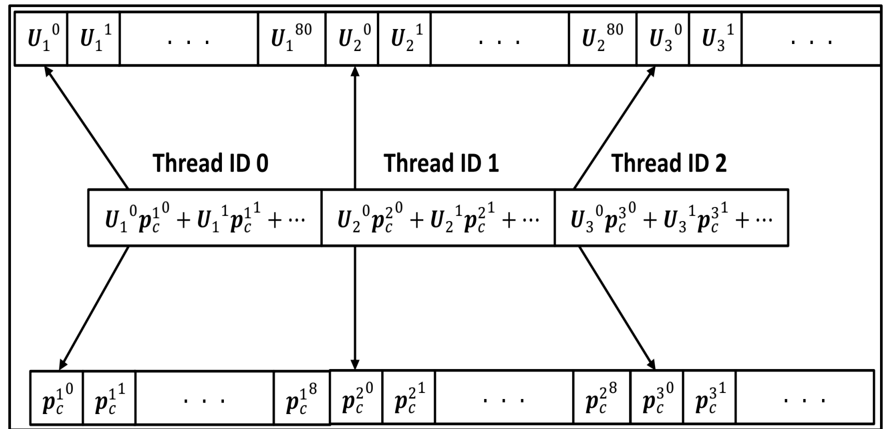


Figure 6. Strided access of the block-matrix U_i and block-vector p_c^i.

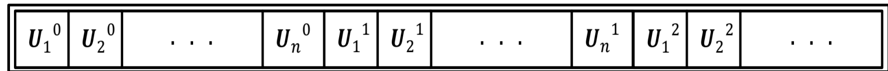


Figure 7. Memory layout of the proposed *continuous-element* layout for block-matrix U_i.

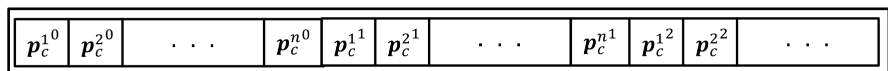


Figure 8. Memory layout of the proposed *continuous-element* layout for block-vector p_c^i.

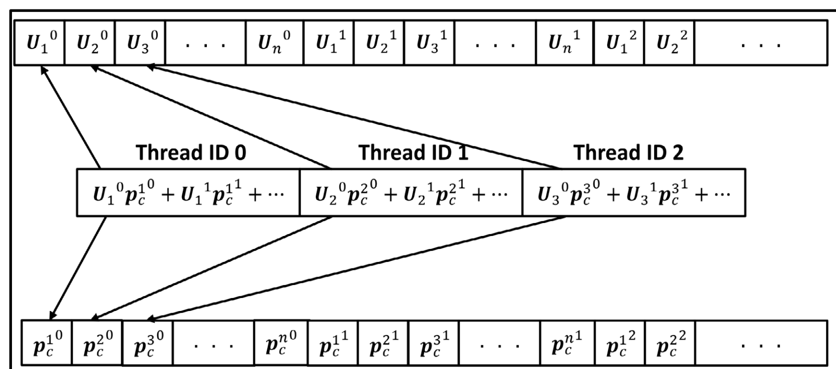


Figure 9. Contiguous access pattern of the block-matrix U_i and block-vector p_c^i.

of memory transactions per request whose functionality is the same across different GPU architectures. As a result, the proposed data layout can be applicable across different GPU architectures without any code changes and improve the performance of the application.

Overall, the studied CUDA kernel optimization and the proposed *continuous-element* data layout can be universally applicable across different GPUs as they affect the fundamental principles which are the same across the different GPUs and are not targeted towards a specific feature of a particular GPU architecture and be constrained to those GPUs only.

6. Data Processing

As mentioned earlier, the use of atomic operations in implementation [1] resulted in precision differences for larger datasets. As a result, in this paper, the atomic operations are eliminated resulting in the requirement of additional initialization data for computing the augmented normal equations and matrix-vector multiplications in the PCG method. In addition, the datasets used are processed to address the non-divergence of a few datasets.

6.1. Additional Initialization Data

Atomic operations were heavily used [1] while computing the augmented normal equation and the matrix-vector multiplication in the PCG method. Using atomic operations allows us to sweep through the entire projections once and compute each of the mathematical variables which are part of the augmented normal equation and the matrix-vector multiplication in the PCG method. While generating a parameter of the augmented normal equation on the GPU, different threads might end up loading from or writing to the same memory. This would result in accessing incorrect data from the memory if the memory is not updated appropriately. Atomic operations are used to avoid reading from or writing to the same memory while other threads in the GPU are using that memory.

As mentioned earlier, computations on GPUs are processed per warp, and the warps can be issued and executed in any order across different runs. For example, in an execution, if a parameter computation on GPU executes warps 1, 5, and 7 in the order, then the next execution may or may not follow the same order. As the algorithm is precision sensitive, this results in a difference in output for each execution. As a result, atomic operations are eliminated from computing the augmented normal equation and the matrix-vector multiplications.

As the atomic operations are eliminated, the parameters cannot be computed by sweeping through the projections. Instead, detailed information about the number of points in a camera, the number of cameras that contain the same point, the start and end index of each camera, and the point with respect to the projections are required during the computation of each parameter in the augmented normal equation and the matrix-vector multiplications.

The datasets [2] provide information about the total number of cameras, points, and projections and the information about parameters of each camera and point. In addition, they contain the camera index, point index, and the observed projection of that camera-point configuration grouped in a single line, as shown in **Figure 10**. In addition, all the cameras for each point are grouped along with the projection information about the point in the camera in consecutive lines.

As the Jacobian are computed for each projection with respect to the camera and point, they are stored in the same order as the observed projections. While computing the point parameters like $\boldsymbol{V}_j$ [4], we need to sweep through all the cameras that contain the point *j* and identify the point Jacobian of that projection. As the current dataset contains all the cameras for each point grouped, we only need the start and the end index of the point as they correspond to the Jacobian stored in the memory. In this paper, we compute the start and end of each point using the prefix sum [18].

In addition, to compute the camera parameters like $\boldsymbol{U}_i$ [4], we need to sweep through all the points in the respective camera. However, as the datasets are grouped in terms of cameras that contain a particular point, we would need to sweep through the entire dataset to identify the camera. This process takes a significant amount of time. As a result, in this paper, we also store the camera and point indexes such that all the points of each camera are grouped. As the camera and point indexes are reordered, we also need to reorder the observed projections. Instead of reordering the observed projections that contain two coordinates for each projection, we generate an index of the projections with respect to points on a single camera being grouped. So, now, the index can be used to point to the respective projection. In addition, similar to the point parameters, we also need to compute the start and end of each camera using the prefix sum.

For example, let us consider we have three points and three cameras, where all three cameras contain all the points. The camera index, point index, and observed projections with respect to the dataset format [2] are shown in **Figure 11**.

$Camera_{Index}$	$Point_{Index}$	$Observed_Projection_{x-axis}$	$Observed_Projection_{y-axis}$

Figure 10. Format of the camera index, point index, and projection coordinates.

	$Camera_{Index}$	$Point_{Index}$	$Observed_Projection_{x-axis}$	$Observed_Projection_{y-axis}$
Line 0:	0	0	x_0	y_0
Line 1:	1	0	x_1	y_1
Line 2:	2	0	x_2	y_2
Line 3:	0	1	x_3	y_3
Line 4:	1	1	x_4	y_4
Line 5:	2	1	x_5	y_5
Line 6:	0	2	x_6	y_6
Line 7:	1	2	x_7	y_7
Line 8:	2	2	x_8	y_8

Figure 11. Format of the camera, point, and projections in the dataset files.

In this paper, we modify the format in the datasets, as in Figure 12, to contain the camera-point indexes with each camera, points grouped, followed by the projection index, and the observed projections, as shown in Figure 13.

In Figure 13, the first set of camera-point indexes in the first and second columns has all the cameras of each point grouped, and the second set of camera-point indexes in the third and fourth columns has all the points of each camera grouped. The fifth column has the projection index with respective to the second set of camera-point indexes. Reading this additional information from the data files is found to be more optimal than computing them in the initialization phase of each execution. In addition, it is also found that computing the start and end indexes for both the cameras grouped and points grouped using prefix sum is time consuming. As a result, the prefix sum values are also stored in the data files, thereby eliminating the need to compute them for every execution.

In addition to the above modifications to the dataset, we also address the non-divergence of a few datasets.

6.2. Dataset Processing

A few of the datasets from [2] are not converging from their initial mean square error. Both the modified BA implementation in [1] and the PBA [11] implementation show similar non-convergent behavior on a few datasets. In a few other datasets that converge, different executions produce different final mean square errors, taking different numbers of PCG iterations. In this paper, we analyze the datasets and propose the removal of a few of the cameras from certain datasets.

From the augmented normal equation, the addition of a regularization term results in a non-singular and positive definite matrix that ensures convergence. As a few of the datasets are not converging, it would mean that the regularization term added was not ensuring convergence due to the matrix being singular.

Cam_{Idx}	Pt_{Index}	Cam_{Idx}	Pt_{Idx}	$Proj_{Idx}$	Obs_Proj_{x-axis}	Obs_Proj_{y-axis}

Figure 12. Modified format of the camera index, point index, and projection coordinates.

	Cam_{Idx}	Pt_{Index}	Cam_{Idx}	Pt_{Idx}	$Proj_{Idx}$	Obs_Proj_{x-axis}	Obs_Proj_{y-axis}
Line 0:	0	0	0	0	0	x_0	y_0
Line 1:	1	0	0	1	3	x_1	y_1
Line 2:	2	0	0	2	6	x_2	y_2
Line 3:	0	1	1	0	1	x_3	y_3
Line 4:	1	1	1	1	4	x_4	y_4
Line 5:	2	1	1	2	7	x_5	y_5
Line 6:	0	2	2	0	2	x_6	y_6
Line 7:	1	2	2	1	5	x_7	y_7
Line 8:	2	2	2	2	8	x_8	y_8

Figure 13. Modified format of the camera, point, and projections in the dataset files.

As the regularization term [2] added is a diagonal matrix, the inverse of each block diagonal matrix in the augmented normal equation can be computed and checked for singularity.

In this paper, we have computed the matrix inverse for all block matrices of augmented normal equations across different camera-point configurations in the datasets [2]. Computing the matrix inverse of each block matrix in the augmented normal equation resulted in a few of the matrices having linearly dependent rows, resulting in a rank deficiency. This leads to the matrices being singular. All the block matrices in the point section of the augmented normal equation $\boldsymbol{V}_{\mu}$ are found to be non-singular. In contrast, few of the block matrices in the camera section of the augmented normal equation $\boldsymbol{U}_{\mu}$ are found to be singular matrices. As a result, specific cameras that result in singular matrices were removed, and the datasets were rearranged accordingly. **Table 1** provides information about each camera that was removed from the datasets and is categorized as per the different locations [2].

After removing the specified cameras in **Table 1**, all the datasets converge from their initial mean square error and produce identical results across different executions.

7. Results and Analysis

The proposed *continuous-element* data layout is implemented on the framework adapted from [1]. The framework has been modified to load the additional initialization data as proposed in this paper. In addition, the proposed data processing has been applied to the datasets provided by the Bundle Adjustments in Large [2], and the modified datasets are used in this paper. Similar to the implementation in [1], the performance results and analysis are provided for ten datasets. The performance study is extended to larger datasets that involve tens of millions of projections. In [1], where few of the datasets were not converging, in this implementation with the data processing, all datasets converge from their initial mean square error. The parameters of the 10 datasets that are used in this paper are provided in **Table 2**, which shows both the original and modified number of cameras, points, and projections.

As in any optimizations, the accuracy of the algorithm is of utmost importance compared to the computational optimizations which can be achieved by changing the algorithm implementation. As a result, in this paper, the initial emphasis is on ensuring that the optimized CUDA code does not alter the convergence of the algorithm. As a result, a detailed analysis of the convergence of the mean square error and the final projections of the optimized CUDA implementation is studied and compared with the naïve CUDA and sequential implementations. Then, the performance results from the optimized CUDA implementation are presented with a comparative analysis of different profiler parameters between optimized and naïve CUDA implementation. Finally, a brief analysis of the complexity of the CUDA optimization is provided.

Table 1. Cameras were removed from datasets of different locations.

Location	Cameras Removed
Dubrovnik	105, 125
Final	724, 460
Trafalgar Square	72
Venice	34, 49, 138, 584

Table 2. Original and modified dataset configurations.

Dataset ID No.	Original Dataset Configurations			Modified Dataset Configurations		
	Number of Cameras	Number of Points	Number of Projections	Number of Cameras	Number of Points	Number of Projections
1	21	11,315	36,455	21	11,315	36,455
2	88	64,298	383,937	88	64,298	383,937
3	182	116,770	668,705	182	116,770	668,705
4	245	198,739	1,091,386	243	198,340	1,084,136
5	744	543,562	3,058,863	741	543,163	3,050,099
6	951	708,276	3,748,892	947	707,877	3,738,748
7	1288	866,452	4,383,006	1284	866,053	4,373,425
8	1936	649,673	5,213,733	1936	649,673	5,213,733
9	4585	1,324,582	9,125,125	4584	1,324,582	9,123,988
10	13,682	4,456,117	28,987,644	13,678	4,455,747	28,975,571

7.1. Accuracy

As mentioned in [1], the algorithm is highly sensitive to floating point precision. A comparison was shown between the accuracy of the sequential and naïve CUDA implementation using the performance parameters described in our previous work [1]. Similarly, we have initially compared the final mean square error across the naïve CUDA and optimized CUDA implementations. The percentage difference in the final mean square error between the naïve and optimized CUDA implementations is evaluated to ascertain the accuracy of the convergence, stability, and reliability of the optimized CUDA implementation. The percentage difference was less than 0.1% for 87 datasets and less than 2.5% for the remaining 8 datasets, indicating that the optimized CUDA has similar convergence to naïve CUDA. Overall, the average percentage error for all 95 datasets is less than 0.22% across different configurations.

Further, similar to the analysis in [1], the stability of the convergence of the optimized CUDA implementation is compared with the naïve CUDA implementation using pixel coordinates. **Figure 14** provides a pictorial representation of the final computed projection and observed projection for the maximum

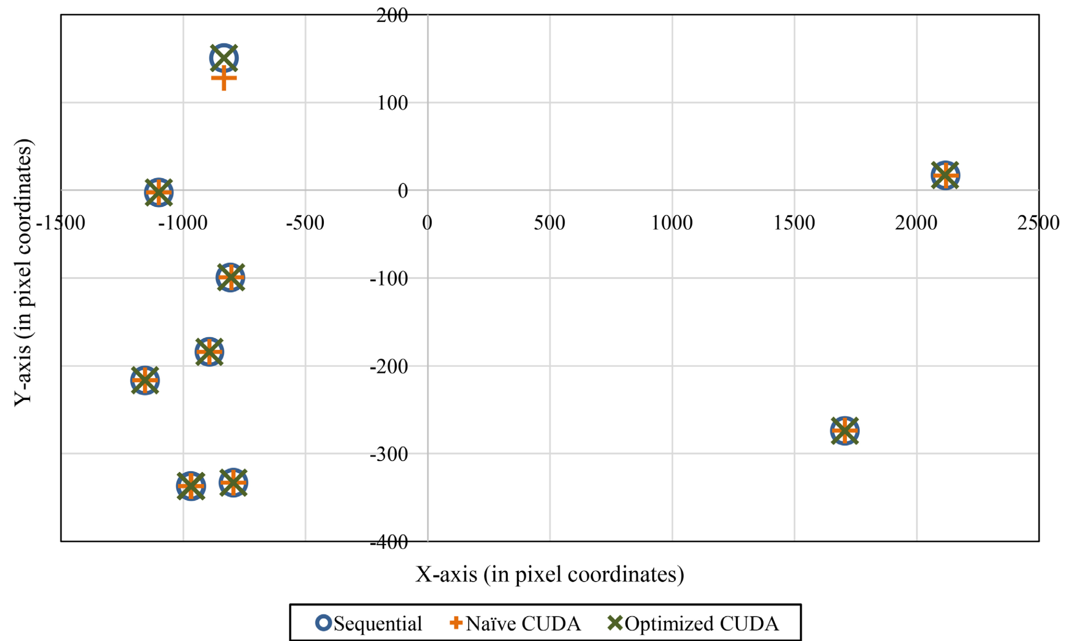


Figure 14. Final projections in pixel coordinates for all sequential, naïve CUDA, and optimized CUDA implementations.

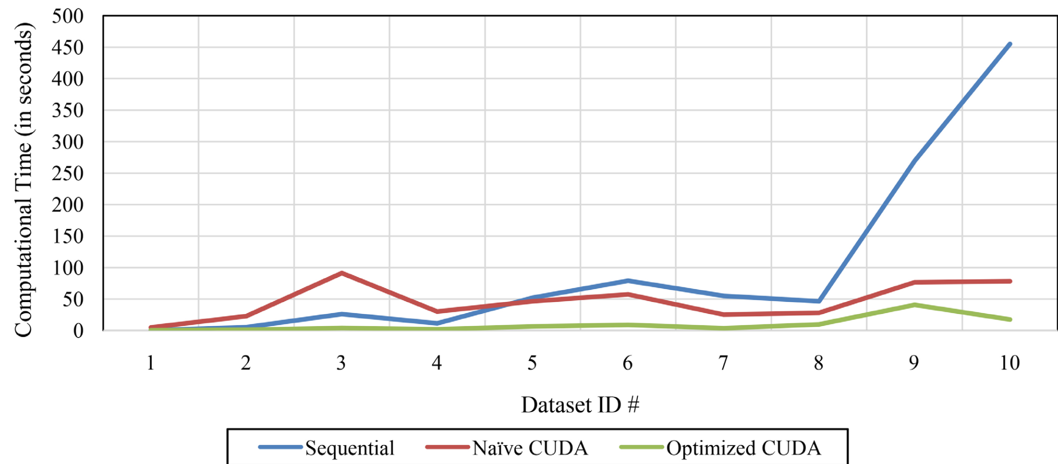
error across the ten datasets. In **Figure 14**, it can be seen that the optimized CUDA implementation converges to the same pixel coordinates as the naïve CUDA and sequential implementations.

From this analysis, it can be concluded that the optimized CUDA implementation does not deteriorate the accuracy of the algorithm and converges to the same pixel coordinates as with the naïve and sequential implementations, indicating that the optimized CUDA implementation of the BA algorithm is stable and reliable.

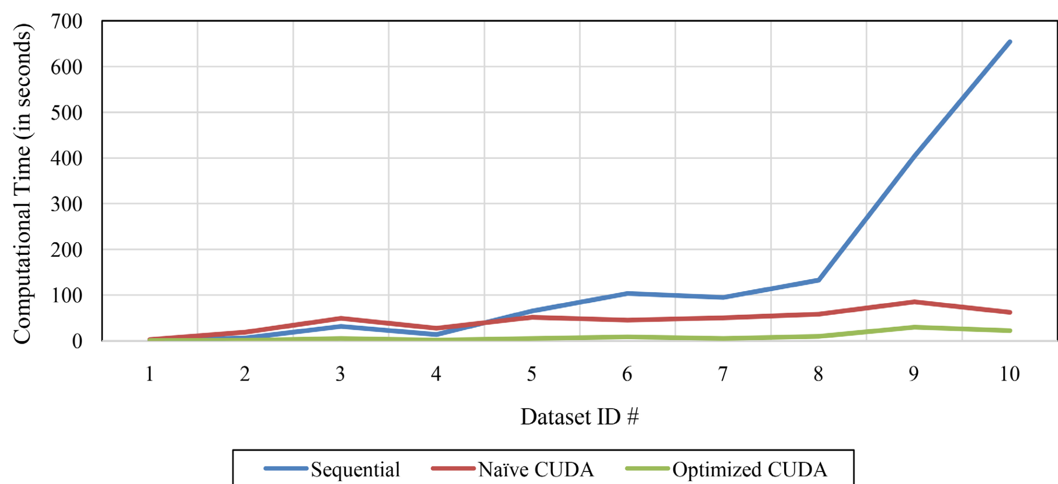
7.2. Performance

Similar to the analysis in [1], we demonstrate the performance studies on both the *with-Schur* and *without-Schur* complement algorithm configurations. In addition, we extended our analysis to understand the impact of computing and using the block-matrix $\boldsymbol{W}$ explicitly and to compute the block-matrix $\boldsymbol{W}$ implicitly by using the camera and point Jacobians in place [1] of the block-matrix $\boldsymbol{W}$ in the different computations. Ongoing in this paper, *Explicit-W* indicates computing block-matrix $\boldsymbol{W}$ explicitly, while *Implicit-W* indicates computing block-matrix $\boldsymbol{W}$ implicitly. The performance study is implemented on a node with a 64-core AMD EPYC 7713P CPU and 3584 cores NVIDIA A30 GPU. The algorithm is executed for a maximum of 50 LM iterations and 100 preconditioned conjugate gradient (PCG) iterations.

Figure 15 shows the computation time of the sequential, naïve CUDA [1] and the optimized CUDA implementation using *Implicit-W* for the ten datasets. In both the *Without-Schur* and *With-Schur* complement, the computation time of



(a)

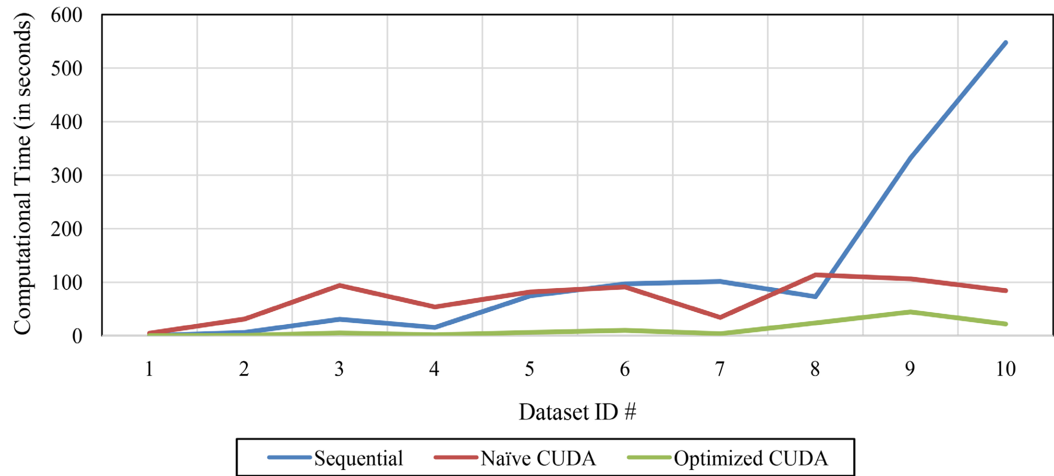


(b)

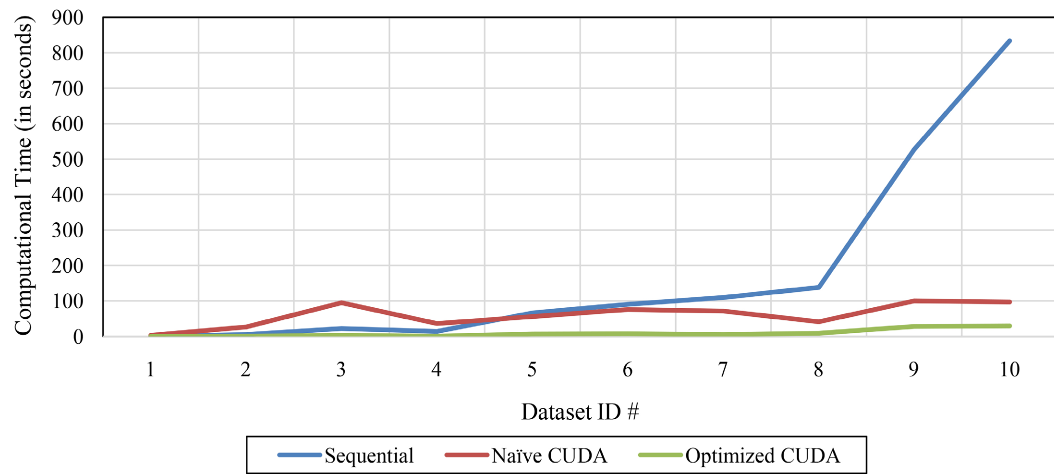
Figure 15. Computational time of sequential, naïve CUDA, and optimized CUDA implementations for *Implicit-W* across different datasets using *without-Schur* and *with-Schur* complement. (a) *Without-Schur* complement, (b) *With-Schur* complement.

the optimized CUDA implementation is significantly less compared to both the sequential and naïve CUDA implementations. At the same time, the computational time of the naïve CUDA implementation is less compared to the sequential implementation only for larger datasets, *i.e.*, for datasets with more than 700 cameras. Furthermore, the computation time of the sequential implementation is significantly increasing for larger dataset sizes. Whereas the computation time for both the CUDA versions increases for larger dataset sizes but not as significantly as with the sequential implementation. This is because the CUDA implementations execute the mathematical operations concurrently.

Figure 16 shows the computation time for the sequential, naïve CUDA and the optimized CUDA implementation using *Explicit-W* for the same 10 datasets. Similar to the *Implicit-W*, the computation time for the optimized CUDA implementation is significantly less compared to both the sequential and naïve



(a)



(b)

Figure 16. Computational time of sequential, naïve CUDA, and optimized CUDA implementations for *Explicit-W* across different datasets using *without-Schur* and *with-Schur* complement. (a) *Without-Schur* Complement, (b) *With-Schur* Complement

CUDA implementations. Also, like *Implicit-W*, the computational time of the naïve CUDA implementation is less compared to the sequential implementation only for larger datasets. Comparing Figure 15 and Figure 16, the sequential implementation is taking significantly higher computation time in *Explicit-W* than the *Implicit-W*. Also, for most of the datasets, it can be observed that the naïve CUDA and optimized CUDA are taking slightly more computation time in the *Explicit-W* configuration. This can be clearly observed in Figure 17, which provides the computational time of all the configurations for the largest dataset with 13,678 cameras.

In Figure 17, it can be observed that the computation time of the *Explicit-W* is more compared to the *Implicit-W* for all the configurations of sequential and CUDA versions. The *Explicit-W* configuration would take more computational time compared to *Implicit-W* because of either having a higher number of PCG

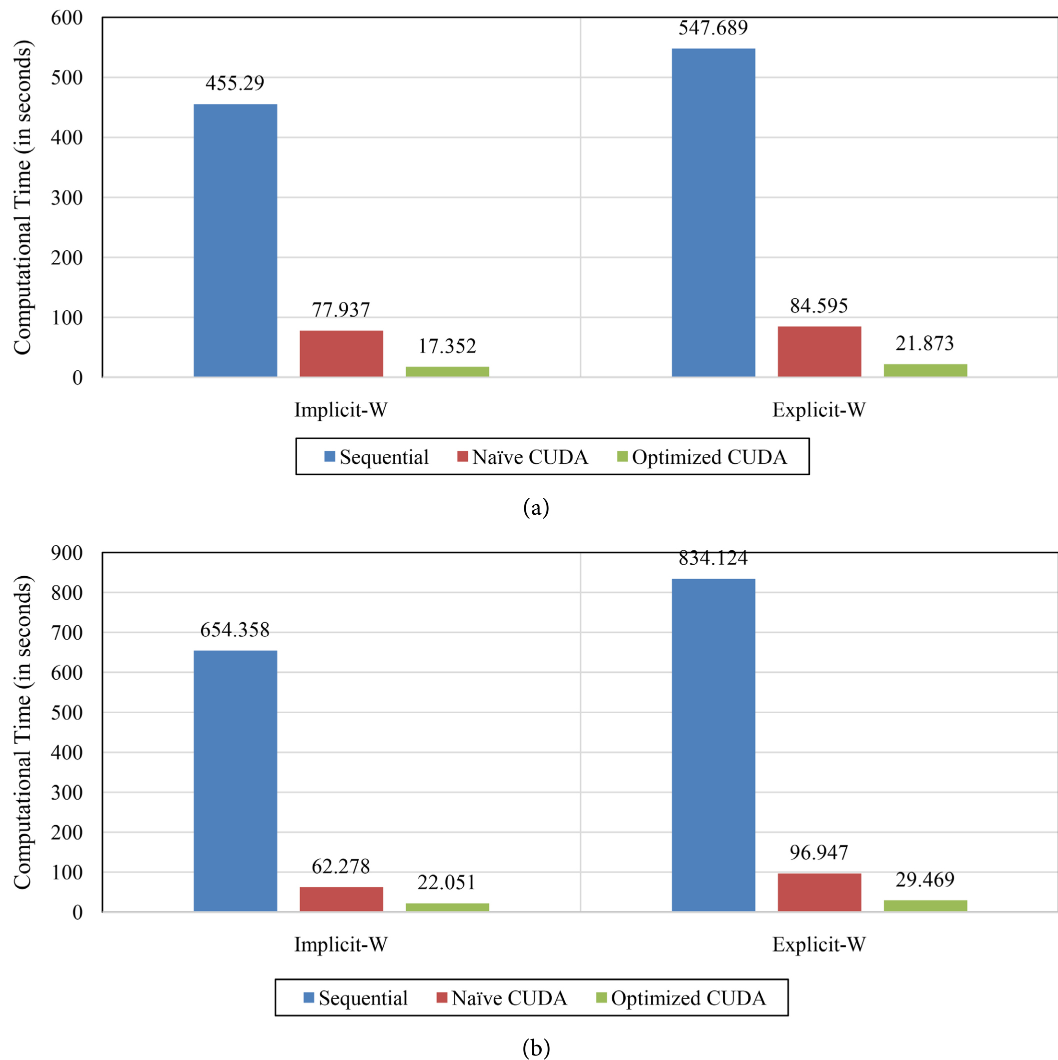


Figure 17. Computational time for the dataset containing 13,678 cameras for all the configurations. (a) *Without-Schur* Complement, (b) *With-Schur* Complement.

iterations or a higher number of total computations per iteration. Furthermore, the computational time of the *Without-Schur* and *With-Schur* complement configurations are random with respect to dataset sizes, as shown in **Figure 15** and **Figure 16**.

In all the implementations, the total number of PCG iterations varies and is based on the convergence of the algorithm. For example, in the dataset with 1936 cameras, the sequential implementation is better compared to the naïve implementation for *Without-Schur* complement and *Explicit-W*, as shown in **Figure 16**. This is because the sequential code has around 506 preconditioned conjugate gradient (PCG) iterations compared to the 1400 iterations by the naïve CUDA implementation. As a result, the naïve implementation has a significantly higher computational time compared to the sequential implementation. Similarly, in **Figure 15** and **Figure 16**, it can be seen that the computational time does not always increase with the increase of the dataset size. This is because

some of the datasets with higher sizes are executing fewer PCG iterations, resulting in fewer overall computations and computational time. The same behavior can be observed in the speedup plot in **Figure 18**.

Figure 18 shows the speedup of the naïve CUDA and optimized CUDA implementation with respect to the sequential implementation. The significant performance improvement achieved by the optimized CUDA implementation in comparison to the naïve CUDA implementation can be seen in the speedup plots. The algorithm achieves the best speedup of approximately 30× on the largest dataset on the *With-Schur Implicit-W* configuration. Overall, the optimized CUDA implementation achieves a speedup of more than 25× on all different configurations of Schur complement and block-matrix $\boldsymbol{W}$ using the largest dataset.

From the speedup plot, it can be seen that the speedup of the optimized CUDA implementation is not steadily increasing with the increase of the dataset size. As mentioned earlier, this is because of the different number of PCG iterations across different implementations or the difference in the overall computations per iteration. As a result, comparing the performance for the complete

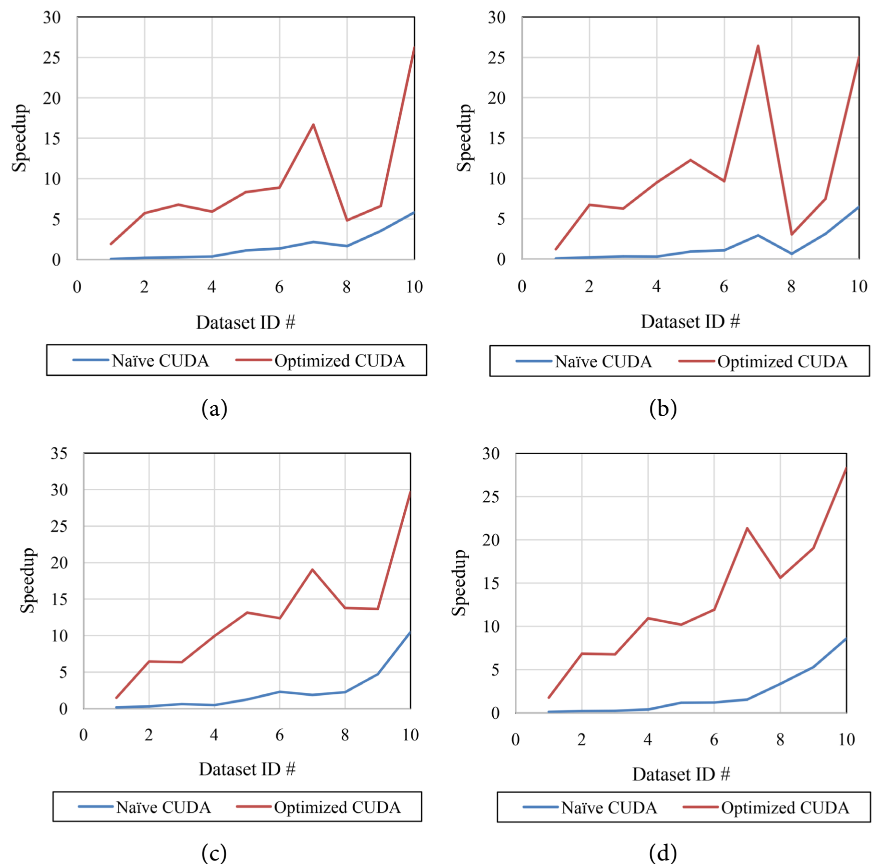


Figure 18. Speedup of the naïve CUDA and optimized CUDA implementations with respect to the sequential implementation with 50 LM and 100 CG iteration. (a) *Without-Schur* Complement & *Implicit-W*, (b) *Without-Schur* Complement & *Explicit-W*, (c) *With-Schur* Complement & *Implicit-W*, (d) *With-Schur* Complement & *Explicit-W*.

convergence of the algorithm for 50 LM steps and 100 PCG iterations does not provide us with sufficient information about the performance as they involve different numbers of total PCG iterations. A common reference for the analysis is required to understand the performance of all the configurations. In this paper, to better understand the performance of different configurations, the total number of LM steps and the total PCG iterations is set to one, and the speedup for all the configurations is evaluated. With this approach, the performance of a configuration can be analyzed based on the total computations per iteration.

Figure 19 shows the speedup of the naïve and optimized CUDA implementations with respect to the sequential implementation with 1 LM and 1 PCG iteration. From the plot, it can be seen that the speedup of the optimized CUDA implementation is monotonically increasing with the size of the datasets. This is because larger datasets contain more computations which can leverage the parallelism offered by the GPUs. The speedup of the naïve CUDA implementation is also increasing monotonically but at a lower rate compared to the optimized CUDA implementation.

As mentioned earlier in **Figures 15-17**, the computational time of the *Explicit-W* configuration is higher compared to the *Implicit-W* configuration for

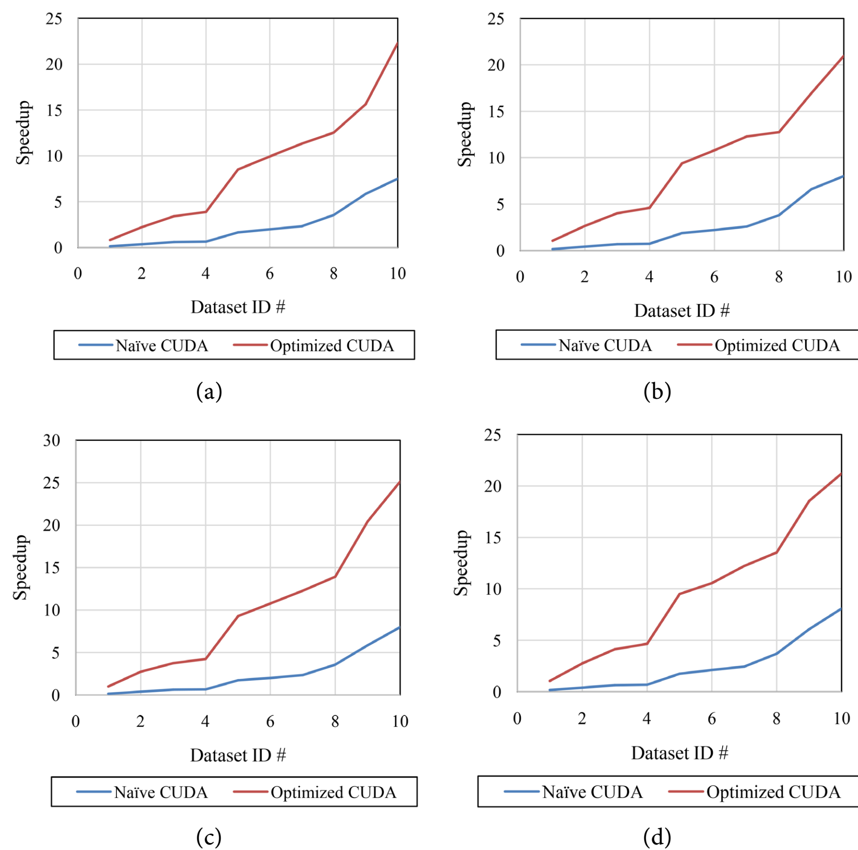


Figure 19. Speedup of the naïve CUDA and optimized CUDA implementations with respect to the sequential implementation with 1 LM and 1 PCG iteration. (a) *Without-Schur* Complement & *Implicit-W*, (b) *Without-Schur* Complement & *Explicit-W*, (c) *With-Schur* Complement & *Implicit-W*, (d) *With-Schur* Complement & *Explicit-W*.

both the *Without-Schur* and *With-Schur* complements. This is due to the *Implicit-W* and *Explicit-W* configurations having a different total number of computations per iteration when using block-matrix $\boldsymbol{W}$. In Table 3, we present the total number of floating-point operations required for a matrix-vector multiplication for both the *Implicit-W* and *Explicit-W* configuration using profiler metrics from the Nsight Compute [19] for the largest dataset. We profiled the code to extract the total number of floating-point operations for the $\boldsymbol{W}_{ij}\boldsymbol{p}_k$ computation [4] in the *Explicit-W* configuration, and for the $\boldsymbol{J}_{ij_c}^{\mathrm{T}}\boldsymbol{J}_{ij_p}\boldsymbol{p}_k$ in the *Implicit-W* configuration for 1 LM and 1 CG iteration.

In Table 3, it can be seen that the *Explicit-W* computation requires more floating-point operations compared to the *Implicit-W* configuration. In addition, we also need to compute the block-matrix $\boldsymbol{W}$ explicitly in the *Explicit-W* configuration. As a result, the *Explicit-W* configuration has more computations per iteration and would increase further with an increase in the total number of iterations. Overall, *Explicit-W* configuration requires more operations compared to *Implicit-W* configuration and this can also be observed in the overall timings as in Table 4. A detailed study of the computational time for different mathematical operations in the *Implicit-W* and *Explicit-W* configurations for 1 LM and 1 PCG method is conducted and analyzed. Table 4 provides us with the computation time for computing the elements of the augmented normal equation and one iteration of the preconditioned conjugate gradient algorithm for the largest dataset.

In Table 4, it can be seen that the computational time for computing the augmented normal equation and the conjugate gradient algorithm is higher in the *Explicit-W* configuration compared to the *Implicit-W* configuration. In addition, storage of the block-matrix $\boldsymbol{W}$ would require huge memory. As a result, the *Implicit-W* computations provide better performance and memory footprint compared to the *Explicit-W* implementations.

Table 3. Total number of floating-point operations for the CUDA implementations.

Dataset ID No.	Total floating-point operations in *Implicit-W* configuration	Total floating-point operations in *Explicit-W* configuration
1	1,203,015	1312,380
2	12,669,921	13,821,732
3	22,067,265	24,073,380
4	35,776,488	39,028,896
5	100,653,267	109,803,561
6	123,378,684	134,594,928
7	144,323,025	157,443,300
8	172,053,189	187,694,388
9	301,091,604	328,463,568
10	956,193,843	1,043,120,556

Table 4. Computational time (milliseconds) to compute augmented normal equation and PCG algorithm.

Configuration	Optimized CUDA Implementation	
	Augmented Normal Equation	PCG Algorithm
1	50	70
2	135	74
3	49	55
4	135	57

In Figure 19, it can also be seen that the performance of the optimized CUDA implementation is better compared to the naïve CUDA. One of the main differences between the naïve and the optimized CUDA implementations is the *continuous-element* data layout. We analyze the impact of the proposed data layout on the performance of the algorithm using the Nsight Compute [19] profiler. The profiler is used to extract performance metrics of the load and store functionality of the GPUs.

Primarily, we have extracted the global load and store efficiency metrics that provide a ratio of the requested global memory throughput to the required global memory throughput for both the load and store operations. In addition, the global load and store transactions per request metrics are also extracted, which provides information about the total number of load and store memory transactions required to fulfill the data request. Higher load and store efficiencies closer to 100% and lower transactions per request closer to 1 imply better memory performance. We have extracted the metrics mentioned above from the CUDA kernel that computes both the camera and point Jacobians. This CUDA kernel is used for illustration of the metrics as it involves a significantly higher number of floating-point operations compared to most of the other CUDA kernels. Table 5 provides information on the metrics for the largest dataset.

From the table, it can be seen that the global load and store transactions per request have improved in the optimized CUDA implementation compared to the naïve CUDA implementation. In fact, the global store transactions have significantly improved from around 30 transactions to around 5 transactions per request. This reduction in the transactions per request will greatly reduce the total load and store transactions required by the kernel. Similarly, improvement is also evident in the global load and store efficiencies, where the global store efficiency has improved to more than 80% compared to the 13.33% in the naïve CUDA implementation. Improved efficiency shows that a majority of the transactions are being utilized effectively. These improvements in the transactions per request and efficiency have greatly improved the overall performance of the algorithm.

Table 5. Profiler metrics of the jacobian computation.

Dataset ID No.	Naïve CUDA Implementation				Optimized CUDA Implementation			
	Global Load Efficiency	Global Load Transactions per Request	Global Store Efficiency	Global Store Transactions per Request	Global Load Efficiency	Global Load Transactions per Request	Global Store Efficiency	Global Store Transactions per Request
1	31.21	4.82	13.33	29.98	52.75	2.83	82.75	4.83
2	28.16	8.47	13.33	30	41.57	5.74	82.76	4.83
3	28.14	8.58	13.33	30	38.71	6.24	82.76	4.83
4	28.18	8.54	13.33	30	38.85	6.19	100	4
5	28.07	8.79	13.33	30	34.08	7.24	82.76	4.83
6	28.22	8.45	13.33	30	33.83	7.07	88.89	4.5
7	28.32	8.26	13.33	30	33.77	6.92	82.76	4.83
8	27.47	10.41	13.33	30	32.65	8.65	82.76	4.83
9	27.81	9.34	13.33	30	30.01	8.47	88.89	4.5
10	27.68	9.85	13.33	30	29.96	8.92	82.76	4.83

7.3. Complexity

Further, we analyze the complexity of the BA algorithm. The BA algorithm involves many sparse matrix-vector and dense vector-vector operations. As a result, the algorithm's computational complexity is not straightforward and is a combination of different mathematical operations. As a result, we made an effort to analyze the computational complexity of the BA algorithm using the computational time and the different input parameters. The code is implemented for a single LM and PCG iteration so that the total number of LM steps and the conjugate iterations are constant across all datasets. Also, the total number of camera parameters and the 3D points are constant across all datasets.

Each dataset has a different number of images, points, and projections. A smaller number of images and points can have a larger number of projections and vice versa. As a result, the number of images and points alone would not provide complete information about the complexity of the algorithm. Also, the total number of projections is significantly larger compared to the number of images and points. As a result, we have used the number of projections as the parameters to compare the complexity. The optimized CUDA implementation that proposes a memory data layout only alters the location of different elements in the memory and does not alter the algorithm implementation. Similarly, the CUDA kernel optimization does not change the algorithm implementation but distributes and executes the algorithm in parallel. As a result, the complexity of the algorithm does not change between the naïve and optimized CUDA implementation in terms of the algorithm execution. However, it does change in terms of the time complexity between the naïve and optimized CUDA implementation.

In Figure 20, the trendline equations of the projections versus computational times of the naïve and optimized CUDA implementations are shown. The slope

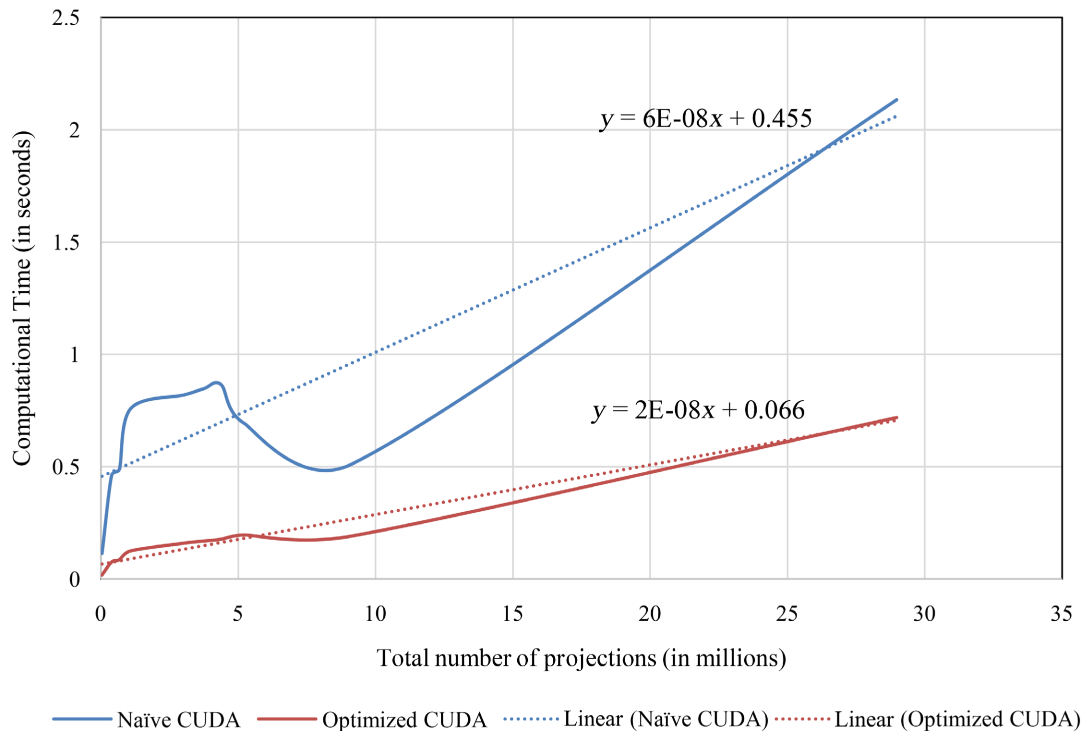


Figure 20. Trendline equations of projections versus computation time of naïve and optimized CUDA.

of the optimized CUDA implementation trendline is approximately 3× of the naïve implementation, correlating with the 30× speedup achieved. The difference in the slope demonstrates the decrease in time complexity for the optimized CUDA implementation.

8. Conclusion and Future Work

In this paper, we proposed a new memory data layout that has improved the memory throughput and reduced the total computational time of the algorithm. We have demonstrated the performance benefits of the proposed data layout through the profiler metrics. The proposed data layout can also be adapted across the other state-of-the-art implementations and would improve their performances. This has been illustrated by implementing the proposed data layout on the framework from [1], and the timing profile has shown a performance improvement. The proposed data layout can be used in applications in other domains where the applications benefit from the spatial locality in the memory.

In addition, we have also studied the impact of computing the block-matrix $\boldsymbol{W}$ explicitly and implicitly by using the camera and point Jacobians in place of the block-matrix $\boldsymbol{W}$. From the study, it is evident that the *Implicit-W* configuration takes less computational time compared to the *Explicit-W* configuration. Also, we have optimized a few of the CUDA kernels by distributing the computations related to camera sections across blocks and the point iterations across threads generating intermediate results that are aggregated by a single thread in the end. We have also preprocessed the datasets such that all the datasets con-

verge from their initial mean square error. Overall, the CUDA implementation with the proposed data layout and the optimizations has achieved a speedup of approximately 30× for the largest dataset, which has 13,678 cameras, 4,455,747 points, and 28,975,571 projections. Overall, a speedup of more than 25× has been achieved with all of the configurations.

The proposed data layout emphasizes on improving the spatial locality of the data in memory, where the consecutive memory is accessed significantly. This deteriorates the temporal locality of the memory, by using the same set of memory repetitively. In the BA algorithm improving spatial locality with deteriorating temporal locality still improved the overall performance of the algorithm.

In addition to the performance optimizations presented in the paper, further studies on the use of complex preconditioners to reduce the total number of conjugate gradient iterations should be researched. The increased computations from generating and utilizing the complex preconditioner can be addressed by asynchronously implementing the computations on the GPUs. In addition, the proposed CUDA optimizations would further improve the computational time in using the complex preconditioners.

Acknowledgements

We want to thank NVIDIA Corporation for providing access to the GPU cluster for development purposes.

We would also like to thank Sameer Agarwal *et al.* for providing the datasets and additional information through the online URL https://grail.cs.washington.edu/projects/bal/ and Changchang Wu *et al.* for releasing the software and additional information through the online URL https://grail.cs.washington.edu/projects/mcba/.

Conflicts of Interest

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

References

[1] 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

[2] Agarwal, S., Snavely, N., Seitz, S.M. and Szeliski, R. (2010) Bundle Adjustment in the Large. In *European Conference on Computer Vision*, Springer, Berlin, Heidelberg, 29-42. https://doi.org/10.1007/978-3-642-15552-9_3

[3] Choudhary, S., Gupta, S. and Narayanan, P.J. (2010) Practical Time Bundle Adjustment for 3d Reconstruction on the GPU. In *European Conference on Computer Vision*, Springer, Berlin, Heidelberg, 423-435. https://doi.org/10.1007/978-3-642-35740-4_33

[4] Lourakis, M.I.A. and Argyros, A.A. (2009) SBA: A Software Package for Generic Sparse Bundle Adjustment. *ACM Transactions on Mathematical Software* (*TOMS*), **36**, 2. https://doi.org/10.1145/1486525.1486527

[5] Tomov, S., Dongarra, J., Volkov, V. and Demmel, J. (2009) MAGMA Library. University of Tennessee and University of California, Knoxville, TN, and Berkeley, CA. https://icl.utk.edu/magma/

[6] Agarwal, S. and Mierle, K. (2012) Ceres Solver. http://ceres-solver.org/

[7] https://developer.apple.com/documentation/accelerate

[8] https://eigen.tuxfamily.org/dox/group__TopicSparseSystems.html

[9] Byröd, M. and Åström, K. (2009) Bundle Adjustment Using Conjugate Gradients with Multiscale Preconditioning. 7-10 September 2009, *British Machine Vision Conference*, *BMVC* 2009, London, 1-10.

[10] Byröd, M. and Åström, K. (2010) Conjugate Gradient Bundle Adjustment. *European Conference on Computer Vision*, Springer, Berlin, Heidelberg, 114-127. https://doi.org/10.1007/978-3-642-15552-9_9

[11] Wu, C.C., Sameer, A., Brian, C. and Seitz, S.M. (2011) Multicore Bundle Adjustment. In *Computer Vision and Pattern Recognition* (*CVPR*), 20-25 June 2011, Colorado Springs, 3057-3064. https://doi.org/10.1109/CVPR.2011.5995552

[12] Zheng, M.T. Zhou, S.P., Xiong, X.D. and Zhu, J.F. (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

[13] https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__TEXREF__DEPRECATED.html

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

[15] https://technical.city/en/video/A30-PCIe-vs-H100-PCIe

[16] https://www.nvidia.com/en-us/data-center/h100/

[17] https://docs.nvidia.com/cuda/cublas/index.html

[18] Blelloch, G.E. (1990) Prefix Sums and Their Applications.

[19] https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html