MegBA: A GPU-Based Distributed Library for Large-Scale Bundle Adjustment

Jie Ren[1,2], Wenteng Liang[1], Ran Yan[1](✉), Luo Mai[2], Shiwen Liu[1], and Xiao Liu[1]

[1] Megvii Inc., Beijing, China
yanran2012@gmail.com
[2] The University of Edinburgh, Edinburgh, UK

Abstract. Large-scale Bundle Adjustment (BA) requires massive memory and computation resources which are difficult to be fulfilled by existing BA libraries. In this paper, we propose MegBA, a GPU-based distributed BA library. MegBA can provide massive aggregated memory by automatically partitioning large BA problems, and assigning the solvers of sub-problems to parallel nodes. The parallel solvers adopt distributed Precondition Conjugate Gradient and distributed Schur Elimination, so that an effective solution, which can match the precision of those computed by a single node, can be efficiently computed. To accelerate BA computation, we implement end-to-end BA computation using high-performance primitives available on commodity GPUs. MegBA exposes easy-to-use APIs that are compatible with existing popular BA libraries. Experiments show that MegBA can significantly outperform state-of-the-art BA libraries: Ceres (41.45×), RootBA (64.576×) and DeepLM (6.769×) in several large-scale BA benchmarks. The code of MegBA is available at: https://github.com/MegviiRobot/MegBA.

1 Introduction

Bundle Adjustment (BA) is the foundation for many real-world 3D vision applications [20,31], including structure-from-motion and simultaneous-localization-and-mapping. A BA problem minimises the re-projection error between camera poses and map points. The error is a non-linear square function, and it is minimised through iterative methods, such as Gauss-Newton (GN) [34], Leverberg-Marquardt (LM) [25] and Dog-Leg [29]. In each iteration, a BA library differentiates the errors with respect to solving states and constructs a linear system

J. Ren and W. Liang—Equal contribution, work was done during their internship in Megvii Inc.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-19836-6_40.

S. Avidan et al. (Eds.): ECCV 2022, LNCS 13697, pp. 715–731, 2022.
https://doi.org/10.1007/978-3-031-19836-6_40

which is solved by optimisation algorithms, such as Cholesky decomposition [4] and Precondition Conjugate Gradient (PCG) [9].

Large-scale BA is increasingly important given the recent rise of city-level high-definition maps for autonomous driving [3,21,22,24] and indoor maps for augmentation reality [28,33,38]. A structure-from-motion application, for example, can produce massive images [2,15], resulting in billions of points and observations to be adjusted. Such a BA problem is orders of magnitude larger than those in conventional vision applications [32,39].

Existing BA libraries (e.g., g2o [12] and Ceres [1]) however provide insufficient support for large-scale BA. We observe several reasons: **(i)** Existing libraries focus on single-node execution, and they lack algorithms to distribute computation. They thus cannot provide massive aggregated memory that is the key for large-scale BA. Even though there are algorithms, such as RPBA [26], DPBA [6] and STBA [39], which explore distributed BA. These algorithms adopt *approximation* which can adversely affect the precision of found solutions. **(ii)** Existing BA libraries are designed for CPU architectures, and they under-utilise GPUs which is particularly useful for large-scale BA. Even though there are systems, such as PBA [36], to accelerate BA with GPUs. They leave key BA operations un-accelerated (e.g., error differentiation and linear system construction). DeepLM [16] offloads error differentiation into GPUs through PyTorch, but the performance is often sub-optimised.

In this paper, we propose MegBA, a novel GPU-optimised distributed library for large-scale BA. The design of MegBA makes several contributions:
(1) Distributed BA algorithms. MegBA provides a large amount of aggregated memory by distributing BA computation to multiple nodes. To this end, we propose a generic BA problem partitioning method. This method leverages a key observation in BA problems: BA problems are often expressed as graphs where nodes represent points/cameras, and edges represent the associations between cameras and points. MegBA can thus automatically partition the graphs based on edges, and ensure each sub-graph has an equal number of edges (with an aim of achieving load balancing). MegBA further assigns sub-graphs to distributed nodes and merges the local solutions to sub-graphs. To ensure that distributed BA can offer the precision as those computed by single-node BA libraries, we propose the distributed PCG algorithm and the distributed Schur elimination algorithm. These two algorithms synchronise the states of solvers on parallel nodes, and the synchronisation is realised using NCCL.
(2) GPU-Optimised BA computation. MegBA thoroughly optimise BA computation for GPUs, thus providing massive computation power for large-scale BA. Computation-intensive operators (e.g., inverse, inner project, etc.) are implemented as Single-Instruction-Multiple-Data (SIMD) operators. MegBA store data in *JetVector*, a data structure that stores BA data in SIMD-friendly vectors, and JetVector minimises data serialisation cost between CPUs and GPUs. To minimise data movement cost which could block GPU execution, MegBA has algorithms that can predict the GPU memory usage of BA, thus pre-fetching BA data if possible. It exposes easy-to-use APIs that are compatible with g2o and Ceres. Ceres and g2o applications can be thus easily ported to MegBA.

We evaluate the performance of MegBA on servers and each server has 8 NVIDIA V100 GPUs. Experiments with public large BA datasets (i.e., Final-13682 [2]) show that MegBA can out-perform Ceres by up to 41.45×, and RootBA [7] by up to 64.576×, indicating the benefits of optimising BA computation for GPUs. We further compare MegBA with DeepLM [16], a GPU-based BA library. MegBA out-performs DeepLM by 5.213× on 4 GPUs. With 8 GPUs, MegBA out-performs DeepLM by 6.769×, making MegBA the state-of-the-art BA library on GPUs.

To evaluate the scalability of MegBA, we construct an extremely large synthetic BA dataset which is modelled after by the BA problems we have in real-world applications. This dataset contains 80 million observations, 2.76× larger than Final-13682. DeepLM and RootBA incur out-of-memory error and cannot handle such a dataset. On the contrary, MegBA can solve this BA problem in 216.26 s by distributing BA computation to 8 GPUs, which is 23.54× faster than Ceres.

2 Related Work

This section describes the related work of MegBA. g2o [12] and Ceres [1] are **exact BA libraries** that can compute high-accuracy solutions to BA problems. These libraries are designed for using parallel CPU cores, and they cannot use GPUs. These libraries also fail to provide distributed execution, which makes them suffer from out-of-memory issues in solving large-scale BA problems.

Approximated BA algorithms can substantially speed up BA, though often come with a compromise in the quality of BA solutions. PBA [36] is limited to run on a single device. $\sqrt{BA}$ [8] replaced Schur Complement with a memory-efficient nullspace projection of Jacobian, thus improving its performance with single-precision float numbers. iSAM [18] and iSAM2 [17] exploit states ordering; while ICE-BA [23] exploits the states in temporal orders. Though fast in speed, approximated BA algorithms modify the original BA problems, which adversely affect the quality of BA solutions. As a result, commercial 3D vision software, such as PIX4D[1] usually avoid any form of approximation and adopt exact BA libraries if possible.[1]

Distributed BA libraries have been recently designed for large-scale BA. Anders Eriksson et al. [10] present consensus-based optimisation which leverages proximal splitting. Runze Zhang et al. [37] purpose an Alternating Direction Method of Multipliers to distribute the optimisation problem. Later RPBA [26], DPBA [6], STBA [39] partition the BA problems based on ADMM. These ADMM-based systems incur massive redundant computation on distributed devices, making them sometimes under-perform single-node libraries. Further, their users must manually partition BA problems, resulting in sub-optimal distributed performance. BA-Net [30] and DeepLM [16] leverages GPUs to speed up BA. They however rely on PyTorch to use GPU, which incurs non-trivial performance overheads when using GPU and extra memory copies. Decentralised SLAM libraries, such as DEDV-SLAM [5], often solve approximated BA problems on distributed robots, then they merge local solutions. However, the merged solution is not equivalent to the original global BA problem.

[1] https://www.pix4d.com/.

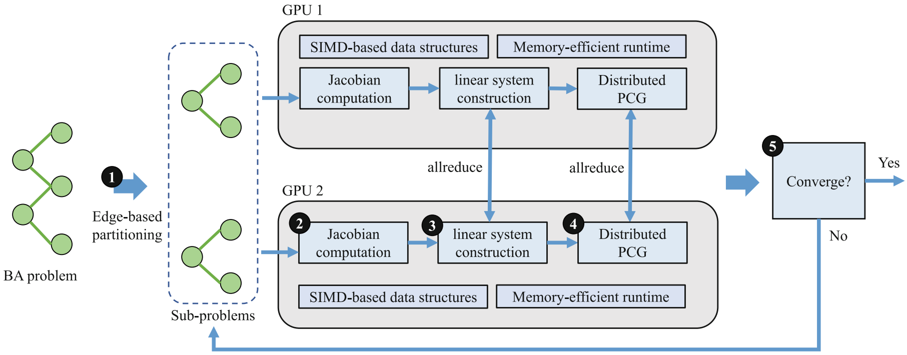


Fig. 1. MegBA overview. ❶ *MegBA partitions a BA problem based on edges. BA sub-problems are in the same size, and they are dispatched to distributed GPUs. Each GPU computes Jacobians* ❷*, constructs a linear system* ❸*, and solve the linear system using the distributed PCG algorithm* ❹*. The communication involved in linear system construction and distributed PCG is implemented using allreduce operations. Step* ❷*,* ❸*, and* ❹ *are executed iteratively until* ❺ *convergence criteria has been met.*

Custom hardware and algorithms are useful in accelerating BA [14]. GBP [27] uses a neural processing unit (i.e., GraphCore IPU) to speed up BA; but the limited availability of IPU makes GBP difficult to be used as a general solution. Practitioners also propose an approximated BA solver tailored for facial capture [11], and this solver cannot be used for arbitrary BA problems such as structure-from-motion.

3 Preliminaries

This section introduces the preliminaries of MegBA. A BA problem can be expressed as a graph, and its solving is realised an iterative process which minimises a non-linear square error objective function:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \sum \boldsymbol{e}_{i,j}^\top \boldsymbol{\Sigma}_{i,j} \boldsymbol{e}_{i,j}, \tag{1}$$

where $\boldsymbol{e}_{i,j}$ is the constraint (i.e. error or graph edge) between state (i.e. parameters or graph nodes) $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, $\boldsymbol{\Sigma}_{i,j}$ is an information matrix.

Solving Eq. 1 is equivalent to iteratively updating the incremental amount $\boldsymbol{\Delta x}$, given by the linear system $\boldsymbol{H \Delta x} = \boldsymbol{g}$, upon the current state $\boldsymbol{x}$ until convergence. The Hessian matrix $\boldsymbol{H} = \boldsymbol{J}^T \Sigma \boldsymbol{J}$ for GN method and $\boldsymbol{H} = \boldsymbol{J}^T \Sigma \boldsymbol{J} + \lambda \boldsymbol{I}$ for LM method, the residual vector $\boldsymbol{g}$ equals to $-\boldsymbol{J}^T \Sigma \boldsymbol{r}$, $\boldsymbol{J}$ is the Jacobian of the error $\boldsymbol{e}$ with respect to current state $\boldsymbol{x}$.

To solve BA problems, BA libraries can use Schur Complement (SC):

$$\begin{bmatrix} \boldsymbol{B} & \boldsymbol{E} \\ \boldsymbol{E}^T & \boldsymbol{C} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Delta x}_c \\ \boldsymbol{\Delta x}_p \end{bmatrix} = \begin{bmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{bmatrix} \tag{2}$$

where $\boldsymbol{B}$ and $\boldsymbol{C}$ are block diagonal and they are related to camera-camera and point-point edges, respectively. $\boldsymbol{E}$ refers to edges between camera and point. $\boldsymbol{v}$ and $\boldsymbol{w}$ refer to the residual vectors for camera and point states.

Solving $\boldsymbol{H}\Delta\boldsymbol{x} = \boldsymbol{g}$ is equivalent to compute the incremental update for states related to cameras $\Delta\boldsymbol{x}_c$ by solving an alternative linear system, called Reduced Camera System (RCS)

$$[\boldsymbol{B} - \boldsymbol{E}\boldsymbol{C}^{-1}\boldsymbol{E}^T]\Delta\boldsymbol{x}_c = \boldsymbol{v} - \boldsymbol{E}\boldsymbol{C}^{-1}\boldsymbol{w}, \tag{3}$$

and followed by a substitution $\Delta\boldsymbol{x}_c$ into

$$\Delta\boldsymbol{x}_p = \boldsymbol{C}^{-1}\left(\boldsymbol{w} - \boldsymbol{E}^T\Delta\boldsymbol{x}_c\right), \tag{4}$$

to get the update for 3D map points.

BA libraries solve linear systems using either direct methods or iterative methods. Direct methods, such as Gaussian-Elimination, LU, QR, and Cholesky Decomposition, return optimised solution of $\boldsymbol{x}$ in one pass. They however suffer from $O(n^3)$ time and $O(n^2)$ space complexity, making them only suitable for small-scale BA problems. On the contrary, iterative methods, such as PCG [31], are suitable for large-scale BA problems. Specifically, PCG replaces the explicit computation of $\boldsymbol{E}\boldsymbol{C}^{-1}\boldsymbol{E}^T$ with multiple iterative sparse matrix-vector operations. It reduces the space complexity to $O(n)$, thus saving memory.

4 MegBA Design

This section introduces the design of MegBA. A key design goal of MegBA is to transparently distribute the solving of a given BA problem to multiple nodes, thus addressing the memory wall of a single node.

Figure 1 presents an overview of the distributed execution of MegBA. A MegBA user declares a BA problem as a graph. MegBA can automatically partition the BA problem based on edges with an aim of each BA sub-problem to have an even number of edges ❶. Specifically, each GPU first ❷ computes the Jacobian (i.e. differentiation of the edge for the node), and then ❸ construct the linear system, and finally, ❹ apply PCG to compute the update for adjusting the current BA sub-problem. The PCG intermediate state is synchronised so that MegBA can eventually solve the shared global BA problem. The BA update step is iteratively performed until a user-defined convergence criterion is met ❺. Notably, the BA computation on each GPU is implemented as SIMD operations which can best utilise GPUs (The details of SIMD-friendly data structure and the memory-efficient runtime are given in Sect. 5)

4.1 Edge-Based Partitioning Method

We focus on partitioning the Hessian matrix produced in BA. For example, in a BA dataset with 80M edges, a Hessian matrix $\boldsymbol{H}$ consume over 50G memory, leading to over 99.9% storage to be allocated to $\boldsymbol{E}$ and $\boldsymbol{E}^T$. We want to have a generic method that partitions the Hessian matrix in a BA problem. This method needs to assign each parallel device with a part of the matrix $\boldsymbol{E}$ and $\boldsymbol{E}^T$, preferably in equal sizes. The partitioning needs to guarantee that the global solution merged from local solutions is *equivalent* to the one computed using a single node. This equivalence property is the key to ensuring a high-precision solution found by MegBA.

At the high level, MegBA achieves distributed BA using two major components: (i) a method that can divide a BA problem into sub-problems, and (ii) an algorithm that can coordinate distributed PCG algorithms to solve sub-problems in parallel. Our partitioning method is based on a key observation in BA problems: the non-zero blocks in $\boldsymbol{E}$ and $\boldsymbol{E}^T$ are corresponding with edges, i.e., the i-th row j-th column non-zero block in $\boldsymbol{E}$ is computed by $e_{i,j}$, we can partition edges based on the number of available GPUs, and each GPU only store part of these non-zero blocks (We provide an example to illustrate the partitioning process in the supplementary materials).

Assume there are K GPUs, given a BA problem, we tile edges to a vector $\boldsymbol{e} = [\dots e_{i,j} \dots]^T$, then we partition it to several blocks $\boldsymbol{e} = [\boldsymbol{e}_1^T \ \boldsymbol{e}_2^T \dots \boldsymbol{e}_K^T]^T$. The Jacobian $\boldsymbol{J}$ could be partitioned into several blocks:

$$\boldsymbol{J} = \frac{d\boldsymbol{e}}{d\boldsymbol{x}} = \left[\frac{d\boldsymbol{e}_1}{d\boldsymbol{x}}^T \ \frac{d\boldsymbol{e}_2}{d\boldsymbol{x}}^T \dots \frac{d\boldsymbol{e}_K}{d\boldsymbol{x}}^T \right]^T = \left[\boldsymbol{J}_1^T \ \boldsymbol{J}_2^T \dots \boldsymbol{J}_K^T \right]^T. \tag{5}$$

Assuming identity information matrix is given here, Hessian $\boldsymbol{H}$ can be partitioned:

$$\boldsymbol{H} = \boldsymbol{J}^T \boldsymbol{J} \quad = \sum_{k=1}^{K} \boldsymbol{J}_k^T \boldsymbol{J}_k = \sum_{k=1}^{K} \boldsymbol{H}_k. \tag{6}$$

To perform Schur elimination, we represented $\boldsymbol{H}_k$ as sub-blocks:

$$\boldsymbol{H}_k = \begin{bmatrix} \boldsymbol{B}_k & \boldsymbol{E}_k \\ \boldsymbol{E}_k^T & \boldsymbol{C}_k \end{bmatrix}. \tag{7}$$

Matrix blocks in Eq. 2 have the following equivalent forms in the edge-based partition setting: $\boldsymbol{B} = \sum_{k=1}^{K} \boldsymbol{B}_k$, $\boldsymbol{E} = \sum_{k=1}^{K} \boldsymbol{E}_k$, $\boldsymbol{E}^T = \sum_{k=1}^{K} \boldsymbol{E}_k^T$, and $\boldsymbol{C} = \sum_{k=1}^{K} \boldsymbol{C}_k$. The number of non-zero parameter blocks in $\boldsymbol{E}$ or $\boldsymbol{E}^T$ equals the number of edges. Notably, the sub-matrices $\boldsymbol{B}_k$ and $\boldsymbol{C}_k$ have the same number of non-zero elements as $\boldsymbol{B}$ and $\boldsymbol{C}$, respectively. Since we store matrices in the Compressed Sparse Row (CSR) format, each GPU only stores $\frac{1}{K}$ non-zero blocks in $\boldsymbol{E}$ and $\boldsymbol{E}^T$. The blocking strategy greatly alleviates the problem that $\boldsymbol{E}$ and $\boldsymbol{E}^T$ are too large to be stored on a single device.

By applying the above partition method for Eq. 2, an equivalent distributed version can be formulated as follow:

$$\boldsymbol{g} = -\boldsymbol{J}^T \boldsymbol{r} = -[\boldsymbol{J}_1^T \ \boldsymbol{J}_2^T \dots \boldsymbol{J}_K^T][\boldsymbol{r}_1^T \ \boldsymbol{r}_2^T \dots \boldsymbol{r}_K^T]^T = -\sum_{k=1}^{K} \boldsymbol{J}_k^T \boldsymbol{r}. \tag{8}$$

Algorithm 1. Distributed BA

```
Input: BA initial state $\boldsymbol{x} = [\,\boldsymbol{x}_c^T\ \boldsymbol{x}_p^T\,]^T$, vector of edges $\boldsymbol{e}_k$, and local GPU rank $k$
Output: Optimised state $\boldsymbol{x}$
1: while BA Convergence Criteria not satisfied do
2:     $\boldsymbol{r}_k = \boldsymbol{e}_k(\boldsymbol{x}), \boldsymbol{J}_k = d\boldsymbol{e}_k(\boldsymbol{x})/d\boldsymbol{x}$    /* Residual and Jacobian */
3:     $\begin{bmatrix} \boldsymbol{B}_k & \boldsymbol{E}_k \\ \boldsymbol{E}_k^T & \boldsymbol{C}_k \end{bmatrix} = \boldsymbol{J}_k^T\boldsymbol{J}_k, [\,\boldsymbol{v}_k\ \boldsymbol{w}_k\,] = -\boldsymbol{J}_k^T\boldsymbol{r}_k$    /* Hessian and Constant vector */
4:     $\boldsymbol{B} = allreduce(\boldsymbol{B}_k)$, $\boldsymbol{C} = allreduce(\boldsymbol{C}_k)$,
       $\boldsymbol{v} = allreduce(\boldsymbol{v}_k)$, $\boldsymbol{w} = allreduce(\boldsymbol{w}_k)$
           /* $\boldsymbol{B} = \sum_{i=1}^K \boldsymbol{B}_i,\ \boldsymbol{C} = \sum_{i=1}^K \boldsymbol{C}_i,\ \boldsymbol{v} = \sum_{i=1}^K \boldsymbol{v}_i,\ \boldsymbol{w} = \sum_{i=1}^K \boldsymbol{w}_i$ */
5:     $\boldsymbol{\alpha}_k = \boldsymbol{E}_k\boldsymbol{C}^{-1}\boldsymbol{w}$
6:     $\boldsymbol{\alpha} = allreduce(\boldsymbol{\alpha}_k)$    /* $\sum_{i=1}^K \boldsymbol{\alpha}_i$ */
7:     $\boldsymbol{g} = \boldsymbol{v} - \boldsymbol{\alpha}$    /* Constant vector in Equation 3 */
8:     $\Delta\boldsymbol{x}_c = \mathtt{DPCG}(\boldsymbol{0}, \boldsymbol{B}, \boldsymbol{E}_k, \boldsymbol{E}_k^T, \boldsymbol{C}, \boldsymbol{g}, k)$    /* Update $\boldsymbol{x}_c$ using Algorithm 2 */
9:     $\boldsymbol{\beta}_k = \boldsymbol{E}_k^T\Delta\boldsymbol{x}_c$
10:    $\boldsymbol{\beta} = allreduce(\boldsymbol{\beta}_k)$    /* $\sum_{i=1}^K \boldsymbol{\beta}_i$ */
11:    $\Delta\boldsymbol{x}_p = \boldsymbol{C}^{-1}(\boldsymbol{w} - \boldsymbol{\beta})$    /* Increment of $\boldsymbol{x}_p$ */
12:    $\boldsymbol{x}_c = \boldsymbol{x}_c + \Delta\boldsymbol{x}_c$, $\boldsymbol{x}_p = \boldsymbol{x}_p + \Delta\boldsymbol{x}_p$    /* Update state */
13: end while
14: return $\boldsymbol{x} = [\,\boldsymbol{x}_c^T\ \boldsymbol{x}_p^T\,]^T$
```

4.2 Distributed BA Algorithm

By far we have partitioned a BA problem and assigned sub-problems to all GPUs. In the following, we will discuss how does MegBA coordinates the solving of sub-problems in a distributed manner.

Algorithm 1 introduces the overall distributed BA algorithm in MegBA. The distributed BA algorithm takes as initial state and partitioned edges as described in Sect. 4.1. We use *JecVector* to compute the Jacobian and residual (Line 2). *JetVector* is a novel data structure to represent BA data in a SIMD format, it can make full use of the hardware characteristics of GPU (e.g. coalesced memory loading) to do auto-differentiation over millions of edges in parallel. We give a more detailed illustration in Sect. 5.1. Then we build a linear system (Line 3). We perform allreduce on diagonal-blocks and constant vector (Line 4) before solving the linear system because the size of diagonal-blocks and constant vector is small and they would be used several times in the following procedures.

We then compute constant vector in Eq. 3 (Line 5–7) and solve the linear system by using a distributed PCG (DPCG) algorithm (Line 8). Notably, we do necessary allreduce in the DPCG algorithm to guarantee DPCG output the same result as non-distributed PCG solver does in solving Eq. 3, further implementation details will be shown in Sect. 4.3. After solving the linear system in Eq. 3, we compute the increment of $\boldsymbol{x}_p$ following Eq. 4 (Line 9–11). Once we have computed the increment $\Delta\boldsymbol{x}_c$ and $\Delta\boldsymbol{x}_p$, we update the state $\boldsymbol{x}_c$ and $\boldsymbol{x}_p$ (Line 12). If it doesn't satisfy the convergence criteria we will start another loop; otherwise, we will return the optimised state $\boldsymbol{x}$.

Algorithm 2. Distributed PCG (DPCG)

```
Input: Initial state $\boldsymbol{x}^0$, matrix block $\boldsymbol{B}$, $\boldsymbol{E}_k$, $\boldsymbol{E}_k^T$, $\boldsymbol{C}$ of $\boldsymbol{H}_k$, constant vector $\boldsymbol{b}$, and local GPU rank $k$
Output: Solution $\boldsymbol{x}$ for linear system $[\boldsymbol{B} - \boldsymbol{E}\boldsymbol{C}^{-1}\boldsymbol{E}^T]\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{E} = \sum_{i=1}^K \boldsymbol{E}_i$ and $\boldsymbol{E}^T = \sum_{i=1}^K \boldsymbol{E}_i$
$\boldsymbol{r}^0 = \boldsymbol{b} - \mathtt{DSE}(\boldsymbol{x}^0, \boldsymbol{B}, \boldsymbol{E}_k, \boldsymbol{E}_k^T, \boldsymbol{C}^{-1}, k)$        /* Algorithm 3 */
$n = 0$
while Convergence Criteria not satisfied do
    $\boldsymbol{z}^n = \boldsymbol{B}^{-1}\boldsymbol{r}^n$
    $\rho^n = \boldsymbol{r}^{nT}\boldsymbol{z}^n$
    if $n > 1$ then
        $\beta^n = \rho^n/\rho^{n-1}$
        $\boldsymbol{p}^n = \boldsymbol{z}^n + \beta^n\boldsymbol{p}^n$
    else
        $\boldsymbol{p}^n = \boldsymbol{z}^n$
    end if
    $\boldsymbol{q}^n = \mathtt{DSE}(\boldsymbol{p}^n, \boldsymbol{B}, \boldsymbol{E}_k, \boldsymbol{E}_k^T, \boldsymbol{C}^{-1}, k)$        /* Algorithm 3 */
    $\alpha^n = \rho^n/\boldsymbol{p}^{nT}\boldsymbol{q}^n$
    $\boldsymbol{x}^{n+1} = \boldsymbol{x}^n + \alpha^n\boldsymbol{p}^n$
    $\boldsymbol{r}^{n+1} = \boldsymbol{r}^n - \alpha^n\boldsymbol{q}^{n-1}$
    $n = n + 1$
end while
return $\boldsymbol{x}^n$
```

Algorithm 3. Distributed Schur Elimination (DSE)

```
Input: Vector $\boldsymbol{x}$, matrix $\boldsymbol{B}, \boldsymbol{E}_k, \boldsymbol{E}_k^T, \boldsymbol{C}^{-1}$, and local GPU rank $k$
Output: Schur elimination result $[\boldsymbol{B} - \boldsymbol{E}\boldsymbol{C}^{-1}\boldsymbol{E}]\boldsymbol{x}$, where $\boldsymbol{E} = \sum_{i=1}^K \boldsymbol{E}_i, \boldsymbol{E}^T = \sum_{i=1}^K \boldsymbol{E}_i^T$
$\boldsymbol{a}_k = \boldsymbol{E}_k^T\boldsymbol{x}$
$\boldsymbol{a} = allreduce(\boldsymbol{a}_k)$        /* $\sum_{i=1}^K \boldsymbol{a}_i$ */
$\boldsymbol{b} = \boldsymbol{C}^{-1}\boldsymbol{a}$
$\boldsymbol{c}_k = \boldsymbol{E}_k\boldsymbol{b}$
$\boldsymbol{c} = allreduce(\boldsymbol{c}_k)$        /* $\sum_{i=1}^K \boldsymbol{c}_i$ */
$\boldsymbol{d} = \boldsymbol{B}\boldsymbol{x}$
return $\boldsymbol{d} - \boldsymbol{c}$
```

4.3 Distributed PCG

We then discuss how to distribute the PCG algorithm in BA, shown in Algorithm 1. This algorithm first constructs a linear system defined in Eq. 2. It then solves Eq. 3 and computes increment following Eq. 4. It finally uses the increments update state $\boldsymbol{x}$, and tested if a convergence criterion has been met. To guarantee that the distributed BA algorithm achieves the convergence performance, we make Algorithm 1, named DPCG, return the same result as the non-distributed linear solver.

In the following, we describe the execution of DPCG. DPCG takes BA initial state $\boldsymbol{x}^0$, matrix block $\boldsymbol{B}$, $\boldsymbol{E}_k$, $\boldsymbol{E}_k^T$, $\boldsymbol{C}$ of $\boldsymbol{H}_k$, constant vector $\boldsymbol{b}$ as input and output solution $\boldsymbol{x}$ for linear system $[\boldsymbol{B} - \boldsymbol{E}\boldsymbol{C}^{-1}\boldsymbol{E}^T]\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{E} = \sum_{k=1}^{K} \boldsymbol{E}_k$ and $\boldsymbol{E}^T = \sum_{k=1}^{K} \boldsymbol{E}_k$. The procedures of DPCG using Schur elimination is similar to single-node PCG. Notably, the coefficient matrix of the linear system to be solved is Schur complement. The matrix-vector multiplication operations (Line 1, 12 in Algorithm 2) is thus the multiplication between Schur complement and vector. The difference between distributed compared with non-distributed setting is that DPCG only assign $\boldsymbol{E}_k$ and $\boldsymbol{E}_k^T$ rather than the complete matrices $\boldsymbol{E}$ and $\boldsymbol{E}^T$ to GPU k, so we need to guarantee operations that use $\boldsymbol{E}_k$ and $\boldsymbol{E}_k^T$ have the same output compared with using $\boldsymbol{E}$ and $\boldsymbol{E}^T$, these operations happen when doing Schur elimination (Line 1, 12).

Our key idea of computing Schur elimination in a distributed manner is that: the summation of matrix-vector multiplication is the same as the result matrix summation multiplies vector, i.e., $\sum_{k=1}^{K}(\boldsymbol{E}_k\boldsymbol{v}) = \sum_{k=1}^{K}(\boldsymbol{E}_k)\boldsymbol{v}$. We compute an intermediate vector (Line 1) and reduce it (Line 2), then we compute intermediate vectors sequentially (Line 3, 4). We perform all-reduce operation over the intermediate vector (Line 5) and compute another intermediate vector (Line 6. After those procedures, we do subtraction to the last two intermediate vectors and output the final result. The result would be the same as computing the complete Schur complement $[\boldsymbol{B} - \boldsymbol{E}\boldsymbol{C}^{-1}\boldsymbol{E}]$ then multiplying it with vector $\boldsymbol{x}$.

4.4 Complexity Analysis

In the end, we present the complexity analysis of MegBA. Assume that MegBA is given m cameras, n points, and k observations and we often have $k \gg m, n$, the time complexity for building the linear system is $\mathcal{O}(m+n+k)$ and the time complexity for each iteration of the conjugate gradient is $\mathcal{O}(m+n+k)$. Assume we distribute the problem to K GPUs, on each GPU, the time complexity for building the linear system is $\mathcal{O}(m + n + k/K)$ and the time complexity for each iteration of the conjugate gradient is $\mathcal{O}(m+n+k/K)$. The ring all-reduce communication time complexity of each conjugate gradient iteration is $\mathcal{O}(m+n)$. In summary, the total complexity of MegBA is $\mathcal{O}(m+n+k/K)$.

5 MegBA Implementation

This section describes the implementation of MegBA. There are several goals of our implementation: (i) We want to use as many SIMD operations as possible because both computation and memory operations on GPU are essentially SIMD-friendly. (ii) We want to optimise the memory efficiency of MegBA, thus avoiding memory allocation and deallocation; (iii) We want to implement the APIs of MegBA that are fully compatible with existing popular BA libraries: g2o and Ceres. In the following, we highlight how MegBA achieves these implementation goals.

5.1 SIMD-Friendly Data Structures

JetVector is a novel data structure to perform auto-differentiation over millions of edges. Compared to conventional BA data structure: *Jet* implemented in Ceres, *JetVector* represents a list of Jets (i.e., Array-of-Structure) as a single data object where Jet's data fields: *data* and *grad* across all items are represented as single arrays (i.e., Array-within-Structure). When we perform mathematical operations on *JetVector*, we will start as many GPU threads as the elements in it, every GPU thread process one element. Because *data* and *grad* are stored in the structure of Array-within-Structure, the memory transactions are coalesced and make it easy to reach a high memory throughput. The detailed structure layout of *JecVector* could be found in supplementary materials.

Besides *JetVector*, other parts of MegBA are also implemented as SIMD-friendly data structures. The construction of linear system (Line 3 in Algorithm 1) uses L1 cache on GPU to store Jacobian blocks in a SIMD manner. The DPCG algorithm includes a lot of matrix/vector operations which also be benefited from the SIMD structure. A full list of SIMD operations implemented in MegBA can be found in supplementary materials.

5.2 Memory-Efficient Runtime

BA computation involves massive objects to be allocated in GPU memory. To avoid expensive memory allocation [19], we leverage a key observation in BA computation: The automatic differentiation works on GPU buffers that are in the same size across BA iterations. By monitoring the sizes of GPU buffers used in the forward pass of differentiating the BA errors, we can predict the sizes of all memory buffers involved in future BA iterations. Based on this observation, we can pre-allocate these memory buffers in a memory pool, thus avoiding calling the CUDA driver to allocate memory during runtime.

5.3 Easy-to-Use APIs

The APIs of MegBA comprises of two major components:
(i) Declaring BA problems. Following the API convention of g2o and Ceres, a BA problem in MegBA is declared a graph that contains nodes and edges. The MegBA nodes describe the 3D coordinates or the poses of cameras and these nodes can be directly imported from g2o and Ceres applications. The MegBA edges are error functions that can be written using the Eigen library [13], identical to Ceres. A MegBA user can build a large BA problem by adding BA nodes and edges (using the g2o-equivalent *addEdge* and *addNode* functions).
(ii) Choosing BA solvers. MegBA also support users to choose solvers given the characteristics of their BA problems. The default solver is the SIMD-optimised DPCG which can automatically use multiple GPUs. Given a small-scale BA problem where intrinsic parallelism is not sufficient, MegBA provides users with the CPU-optimised CHOLMOD solver [4].

6 Experimental Evaluation

We conduct a comprehensive evaluation with MegBA. The evaluation comprises of BAL [2], 1DSfM [35], and a large synthetic dataset modelled after a city-scale BA application we have in production. The dataset statistic is shown in Table 1. Due to the page limit, this section only presents the results with BAL [2], and we put the results of 1DSfM and the synthetic dataset in the supplementary materials.

We compare MegBA with four baselines: (i) Ceres [1] (version 2.0) is the most popular BA library that can efficiently use massive CPU cores, (ii) g2o [12] is a lightweight CPU-based BA library, (iii) RootBA [7] is a recent CPU-based BA library that uses Nullspace-Marginalization in place of Schur Complement, and (iv) DeepLM [16] is the state-of-the-art GPU-based BA library (2021), and it was shown to out-perform other popular BA libraries: STBA [39] and PBA [36] (We provide comparison results between PBA and MegBA in the supplementary materials).

Table 1. Dataset statistics.

Benchmark	Dataset	#Points	#Observations
BAL	Trafalgar-257	65132	225911
	Ladybug-1723	156502	678718
	Dubrovnik-356	226730	1255268
	Venice-1778	993923	5001946
	Final-13682	4456117	28987644
Synthesised	Synthesised-20000	80000	80000000
1DSfM	Alamo-577	140080	816891
	Ellis_Island-233	9210	20500
	Gendarmenmarkt-704	76964	268747
	Madrid_Metropolis-347	44479	195660
	Montreal_Notre_Dame-459	151876	811757
	Notre_Dame-548	224153	1172145
	NYC_Library-334	54757	211614
	Piazza_del_Popolo-336	29731	150161
	Piccadilly-2292	184475	798085
	Roman_Forum-1067	223844	1031760
	Tower_of_London-484	126648	596690
	Trafalgar-5052	327920	1266102
	Union_Square-816	26430	90668
	Vienna_Cathedral-846	154394	495940
	Yorkminster-429	100426	376980

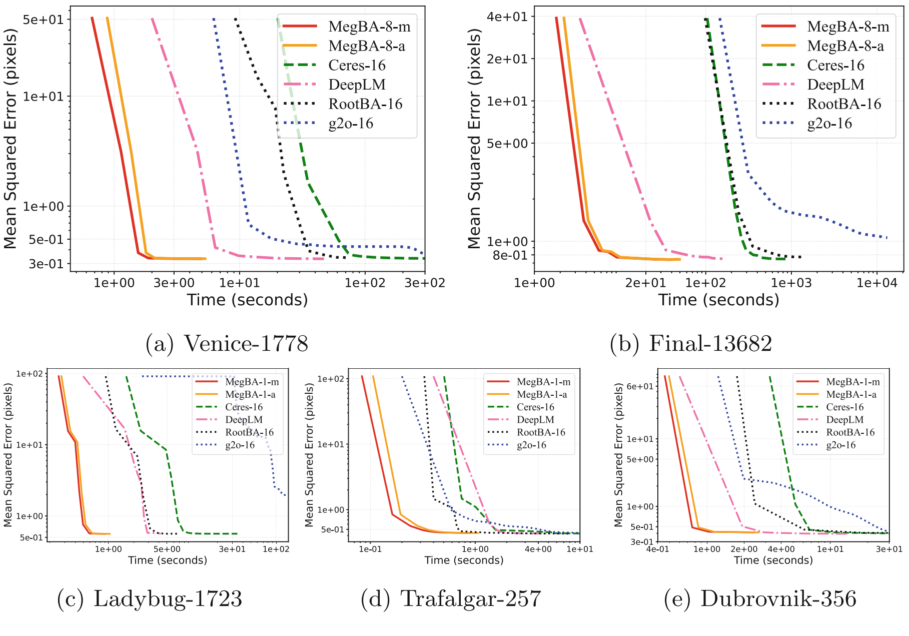
We run experiments on a server that has 80 Intel Xeon 2.5 GHz CPU cores, 8 Nvidia V100 GPUs and 320GB memory. The GPUs are inter-connected using NVLink 2.0. We use 64-bit floating points (FP64) unless otherwise specified.

6.1 Overall Performance

We first evaluate the overall performance of MegBA, Ceres, g2o, RootBA, and DeepLM. MegBA uses from 1 to 8 GPUs, and CPU-based algorithms use 16 threads. We measure the Mean Squared Error (MSE) in pixels over time.

Figure 2 shows the evaluation results. In the Venice-1778 dataset (Fig. 2(a)), MegBA achieves the best performance with 8 GPUs, while DeepLM can only use a single GPU. MegBA completes with 3.34 s while Ceres, RootBA, g2o uses 319.0, 73.94, and 890.6 s, respectively. It shows the substantial speed-up (95.5×, 22.1×, and 266.6×), which indicates the benefits of fully exploiting GPUs to accelerate BA computation. For GPU-based BA libraries, MegBA can complete with 11.96 s while DeepLM spent 24.44 s, showing the effectiveness of implementing full vectorisation for BA on a single GPU. With more GPUs, MegBA out-performs DeepLM by 7.316×, which reflects the necessity of adopting multiple GPUs.

Thanks to the vectorisation and distributed BA designs, MegBA becomes the state-of-the-art in the large BA dataset (i.e., Final-13682). As shown in Fig. 2(b),



(a) Venice-1778

(b) Final-13682

(c) Ladybug-1723

(d) Trafalgar-257

(e) Dubrovnik-356

Fig. 2. Mean Squared Error over Time. *MegBA-X-Y refers to X GPUs while* ***-a*** *refers to auto-differentiation Jacobian and* ***-m*** *refers to analytical differentiation Jacobian. Ceres/RootBA/g2o-X refers to X CPU threads.*

Table 2. Small-scale experiments *We only report the results of MegBA with a GPU because the datasets in this table are small. MSE is the final Mean Squared Error (pixels), Time is BA duration, and Mem is the memory in GB.*

	Trafalgar-257			Ladybug-1723			Dubrovnik-356		
	MSE	Time	Mem	MSE	Time	Mem	MSE	Time	Mem
Ceres-16	0.434	8.160	1.659	0.562	34.50	2.093	0.393	116.0	2.550
DeepLM	0.434	3.820	1.445	0.573	3.930	2.144	0.396	6.119	2.693
g2o-16	0.434	21.69	1.358	1.961	140.7	1.866	0.394	94.39	2.308
RootBA-16	**0.433**	3.307	1.468	0.562	7.050	2.423	0.393	78.16	3.942
MegBA-1-a	0.438	1.364	1.270	0.560	0.932	2.450	0.411	3.640	3.940
MegBA-1-m	0.438	**1.148**	1.010	**0.560**	**0.774**	1.660	0.411	**3.263**	2.480

Table 3. Large-scale experiments.

	Venice-1778			Final-13682		
	MSE	Time	Mem	MSE	Time	Mem
Ceres-16	0.334	319.0	5.983	0.749	916.0	26.08
DeepLM	0.333	24.44	6.256	0.751	149.6	14.89
g2o-16	0.335	890.6	5.999	1.061	13161	36.89
RootBA-16	0.337	73.94	14.14	0.773	1,427	263.2
MegBA-1-a	0.333	11.96	13.68	OOM	OOM	OOM
MegBA-2-a	0.333	7.133	14.51	OOM	OOM	OOM
MegBA-4-a	0.333	4.767	16.76	0.748	28.70	81.03
MegBA-8-a	0.333	3.340	22.61	0.748	22.10	89.74
MegBA-1-m	0.333	10.92	7.870	OOM	OOM	OOM
MegBA-2-m	0.333	6.618	8.693	0.748	50.57	43.60
MegBA-4-m	0.333	4.617	10.95	0.748	26.46	47.33
MegBA-8-m	**0.333**	**3.014**	16.79	**0.748**	**20.68**	56.06

MegBA completes in 22.10 s, while DeepLM uses 149.6 s (6.769× speed-up), Ceres uses 916 s (41.45× speed-up), g2o uses 13161 s (595.5× speed-up), and RootBA uses 1427 s seconds (64.57× speed-up). In other datasets (Fig. 2(c)-(e)), we observe similar speed-up achieved by MegBA, indicating the general effectiveness of our proposed approaches. We omit the discussion of these datasets, and their results are reported in Table 2.

6.2 Scalability

Table 3 further provides the detailed experimental results to show the scalability of MegBA, Ceres and DeepLM. In the Venice-1778 dataset, MegBA can consistently improve its performance by adding GPUs (from 11.96 s to 3.34 s if we increase the number of GPUs from 1 to 8). In addition, the large dataset (Final-

Table 4. Performance with 32-bit and 64-bit floating points.

	Venice-1778			Final-13682		
	MSE	Time	Mem	MSE	Time	Mem
MegBA-1-a(FP32)	0.334	2.620	8.300	OOM	OOM	OOM
MegBA-1-a(FP64)	0.333	11.96	13.68	OOM	OOM	OOM
MegBA-1-m(FP32)	0.333	2.065	4.821	0.750	11.82	24.51
MegBA-1-m(FP64)	0.333	10.92	7.870	OOM	OOM	OOM
MegBA-2-a(FP32)	0.333	1.903	8.447	0.750	11.04	42.48
MegBA-2-a(FP64)	0.333	7.133	14.51	OOM	OOM	OOM
MegBA-2-m(FP32)	0.333	1.353	5.541	0.750	5.133	25.63
MegBA-2-m(FP64)	0.333	6.618	8.693	0.748	50.57	43.60
MegBA-4-a(FP32)	0.333	1.680	10.50	0.749	4.804	45.28
MegBA-4-a(FP64)	0.333	4.767	16.76	0.748	28.70	81.03
MegBA-4-m(FP32)	0.334	1.274	7.598	0.748	**4.279**	28.43
MegBA-4-m(FP64)	0.333	4.617	10.95	0.748	26.46	47.33
MegBA-8-a(FP32)	0.334	1.622	16.02	0.748	8.973	52.15
MegBA-8-a(FP64)	0.333	3.340	22.60	0.748	22.10	89.74
MegBA-8-m(FP32)	**0.333**	**1.271**	12.99	**0.747**	7.582	35.31
MegBA-8-m(FP64)	0.333	3.014	16.79	0.748	20.68	56.06

13682) can better show the scalability of MegBA. By increasing the number of GPUs from 4 to 8, the time can be reduced to 22.10 s.

6.3 Floating Point Precision

The accuracy of solving a BA problem is sensitive to the choice of floating point precision (i.e., 32-bit vs. 64-bit floating points). We further evaluate MegBA in all datasets with 32-bit and 64-bit floating points, and we report the results of Venice-1778 and Final-13682 in Table 4. Other datasets show consistent results and we omit them here. In the dataset of Final-13682, with 4 GPUs, MegBA (FP32) can complete in 4.804 s and MegBA (FP64) can complete in 28.70 s, while both of them are reaching the same MSE. This shows the exactness of the distributed BA algorithm in MegBA. Even with lower precision, MegBA can reach the same MSE as double-precision; but offering 5.97× speed up, making MegBA (FP32) be the state-of-the-art in Final-13682.

7 Conclusion

We present MegBA, a novel GPU-based distributed BA library. MegBA has a set of algorithms that enables automatically distributing BA computation to parallel GPUs. It has a group of SIMD-optimised data structures, and a memory-efficient runtime, making MegBA capable of fully utilising a GPU. MegBA has high-level and compatible APIs, making it quickly become a popular open-sourced BA library. Experimental results show that MegBA can out-perform SOTA BA libraries by orders of magnitudes in several large-scale BA benchmarks.

Acknowledgement. We would like to thank our colleagues at Megvii Inc., Dikai Fan, Shuxue Peng, Yijia He, Zheng Chai, Haotian Zhang and Can Huang for their invaluable inputs. Also, we thank Qingtian Zhu at Peking University and Xingxing Zuo at Technical University of Munich to help with proof reading.

References

1. Agarwal, S., Mierle, K., Others: Ceres solver. http://ceres-solver.org
2. Agarwal, S., Snavely, N., Seitz, S.M., Szeliski, R.: Bundle adjustment in the large. In: European Conference on Computer Vision (2010)
3. Chen, X., Ma, H., Wan, J., Li, B., Xia, T.: Multi-view 3d object detection network for autonomous driving. In: IEEE conference on Computer Vision and Pattern Recognition (2017)
4. Chen, Y., Davis, T.A., Hager, W.W., Rajamanickam, S.: Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate. ACM Trans. Math. Softw. (TOMS) **35**(3), 1–14 (2008)
5. Cieslewski, T., Choudhary, S., Scaramuzza, D.: Data-efficient decentralized visual slam. In: IEEE International Conference on Robotics and Automation (2018)
6. Demmel, N., Gao, M., Laude, E., Wu, T., Cremers, D.: Distributed photometric bundle adjustment. In: IEEE International Conference on 3D Vision (2020)
7. Demmel, N., Sommer, C., Cremers, D., Usenko, V.: Square root bundle adjustment for large-scale reconstruction. In: IEEE Conference on Computer Vision and Pattern Recognition (2021)
8. Demmel, N., Sommer, C., Cremers, D., Usenko, V.: Square root bundle adjustment for large-scale reconstruction. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition (2021)
9. Eisenstat, S.C.: Efficient implementation of a class of preconditioned conjugate gradient methods. SIAM J. Sci. Stat. Comput. **2**(1), 1–4 (1981)
10. Eriksson, A., Bastian, J., Chin, T.J., Isaksson, M.: A consensus-based framework for distributed bundle adjustment. In: IEEE Conference on Computer Vision and Pattern Recognition (2016)
11. Fratarcangeli, M., Bradley, D., Gruber, A., Zoss, G., Beeler, T.: Fast nonlinear least squares optimization of large-scale semi-sparse problems. In: Computer Graphics Forum (2020)
12. Grisetti, G., Kümmerle, R., Strasdat, H., Konolige, K.: g2o: a general framework for (hyper) graph optimization. In: IEEE International Conference on Robotics and Automation (2011)
13. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010)
14. Guo, Y., Liu, J., Li, G., Mai, L., Dong, H.: Fast and flexible human pose estimation with hyperpose. In: Proceedings of the 29th ACM International Conference on Multimedia (2021)
15. Hackel, T., Savinov, N., Ladicky, L., Wegner, J.D., Schindler, K., Pollefeys, M.: Semantic3d. net: a new large-scale point cloud classification benchmark. arXiv preprint arXiv:1704.03847 (2017)

16. Huang, J., Huang, S., Sun, M.: DeepLM: large-scale nonlinear least squares on deep learning frameworks using stochastic domain decomposition. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition (2021)
17. Kaess, M., Johannsson, H., Roberts, R., Ila, V., Leonard, J.J., Dellaert, F.: iSAM2: incremental smoothing and mapping using the bayes tree. Int. J. Robot. Res. **31**(2), 216–235 (2012)
18. Kaess, M., Ranganathan, A., Dellaert, F.: iSAM: incremental smoothing and mapping. IEEE Trans. Robot. **24**(6), 1365–1378 (2008)
19. Koliousis, A., Watcharapichat, P., Weidlich, M., Mai, L., Costa, P., Pietzuch, P.: Crossbow: scaling deep learning with small batch sizes on multi-GPU servers. In: Proceedings of the VLDB Endowment (2019)
20. Konolige, K., Agrawal, M.: FrameSLAM: from bundle adjustment to real-time visual mapping. IEEE Trans. Robot. **24**(5), 1066–1077 (2008)
21. Levinson, J., et al.: Towards fully autonomous driving: systems and algorithms. In: IEEE Intelligent Vehicles Symposium (2011)
22. Li, P., Chen, X., Shen, S.: Stereo r-cnn based 3d object detection for autonomous driving. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition (2019)
23. Liu, H., Chen, M., Zhang, G., Bao, H., Bao, Y.: ICE-BA: Incremental, consistent and efficient bundle adjustment for visual-inertial slam. In: IEEE Conference on Computer Vision and Pattern Recognition (2018)
24. Ma, X., Wang, Z., Li, H., Zhang, P., Ouyang, W., Fan, X.: Accurate monocular 3d object detection via color-embedded 3d reconstruction for autonomous driving. In: IEEE/CVF International Conference on Computer Vision (2019)
25. Marquardt, D.W.: An algorithm for least-squares estimation of nonlinear parameters. J. Soc. Ind. Appl. Math. **11**(2), 431–441 (1963)
26. Mayer, H.: RPBA - robust parallel bundle adjustment based on covariance information (2019)
27. Ortiz, J., Pupilli, M., Leutenegger, S., Davison, A.J.: Bundle adjustment on a graph processor. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition (2020)
28. Park, Y., Lepetit, V., Woo, W.: Multiple 3d object tracking for augmented reality. In: IEEE/ACM International Symposium on Mixed and Augmented Reality (2008)
29. Powell, M.J.: A hybrid method for nonlinear equations. numerical methods for nonlinear algebraic equations (1970)
30. Tang, C., Tan, P.: Ba-net: dense bundle adjustment network. arXiv preprint arXiv:1806.04807 (2018)
31. Triggs, B., McLauchlan, P.F., Hartley, R.I., Fitzgibbon, A.W.: Bundle adjustment-a modern synthesis. In: International workshop on vision algorithms (1999)
32. Vo, M., Narasimhan, S.G., Sheikh, Y.: Spatiotemporal bundle adjustment for dynamic 3d reconstruction. In: IEEE Conference on Computer Vision and Pattern Recognition (2016)
33. Wang, H.-R., Lei, J., Li, A., Wu, Y.-H.: A geometry-based point cloud reduction method for mobile augmented reality system. J. Comput. Sci. Technol. **33**(6), 1164–1177 (2018). https://doi.org/10.1007/s11390-018-1879-3
34. Wedderburn, R.W.: Quasi-likelihood functions, generalized linear models, and the gauss-newton method. Biometrika **61**(3), 439–447 (1974)
35. Wilson, K., Snavely, N.: Robust global translations with 1dsfm. In: Proceedings of the European Conference on Computer Vision (2014)
36. Wu, C., Agarwal, S., Curless, B., Seitz, S.M.: Multicore bundle adjustment. In: IEEE Conference on Computer Vision and Pattern Recognition (2011)

37. Zhang, R., Zhu, S., Fang, T., Quan, L.: Distributed very large scale bundle adjustment by global camera consensus. In: 2017 IEEE International Conference on Computer Vision (2017)
38. Zhao, Y., Guo, T.: Pointar: Efficient lighting estimation for mobile augmented reality. In: European Conference on Computer Vision (2020)
39. Zhou, L., et al.: Stochastic bundle adjustment for efficient and scalable 3d reconstruction. In: European Conference on Computer Vision (2020)