Liu *et al*. *Intell Robot* 2024;4(3):256-75
DOI: 10.20517/ir.2024.17

Intelligence & Robotics

Research Article

Open Access

Parallel implementation for real-time visual SLAM systems based on heterogeneous computing

Han Liu[1,#], Yanchao Dong[2,3,4,#], Chengbin Hou[1], Yuhao Liu[2], Zhanyi Shu[1], Sixiong Xu[2,3], Tingting Lv[2]

[1]CRRC Qingdao Sifang Co., Ltd, R&D Center, Qingdao 266000, Shandong, China.
[2]College of Electronics and Information Engineering, Tongji University, Shanghai 201804, China.
[3]National Key Laboratory of Autonomous Intelligent Unmanned Systems, Shanghai 201210, China.
[4]Frontiers Science Center for Intelligent Autonomous Systems, Ministry of Education, Shanghai 200000, China.
[#]Authors contributed equally.

Correspondence to: Dr. Yuhao Liu, College of Electronics and Information Engineering, Tongji University, No. 4800, Cao'an Highway, Jiading District, Shanghai 201804, China. E-mail: 2211271@tongji.edu.cn

How to cite this article: Liu H, Dong Y, Hou C, Liu Y, Shu Z, Xu S, Lv T. Parallel implementation for real-time visual SLAM systems based on heterogeneous computing. *Intell Robot* 2024;4(3):256-75. http://dx.doi.org/10.20517/ir.2024.17

Received: 24 May 2024 **First Decision:** 9 Jul 2024 **Revised:** 20 Aug 2024 **Accepted:** 26 Aug 2024 **Published:** 31 Aug 2024

Academic Editor: Simon X. Yang **Copy Editor:** Dong-Li Li **Production Editor:** Dong-Li Li

Abstract

Simultaneous localization and mapping has become rapidly developed and plays an indispensable role in intelligent vehicles. However, many state-of-the-art visual simultaneous localization and mapping (VSLAM) frameworks are very time-consuming both in front-end and back-end, especially for large-scale scenes. Nowadays, the increasingly popular use of graphics processors for general-purpose computing, and the progressively mature high-performance programming theory based on compute unified device architecture (CUDA) have given the possibility for large-scale VSLAM to solve the conflict between limited computing power and excessive computing tasks. The paper proposes a full-flow optimal parallelization scheme based on heterogeneous computing to speed up the time-consuming modules in VSLAM. Firstly, a parallel strategy for feature extraction and matching is designed to reduce the time consumption arising from multiple data transfers between devices. Secondly, a bundle adjustment method based solely on CUDA is developed. By fully optimizing memory scheduling and task allocation, a large increase in speed is achieved while maintaining accuracy. Besides, CUDA heterogeneous acceleration is fully utilized for tasks such as error computation and linear system construction in the VSLAM back-end to enhance the operation speed. Our proposed method is tested on numerous public datasets on both computer and embedded sides, respectively. A number of qualitative and quantitative experiments are performed to verify its superiority in terms of speed compared to other states-of-the-art.

Keywords: VSLAM, feature extraction and matching, heterogeneous computing, bundle adjustment

1. INTRODUCTION

With the rapid development of computer vision and the low cost of visual sensors, visual simultaneous localization and mapping (VSLAM) [1–3] methods have been paid special attention for localization and navigation applications such as unmanned cars and automated guided vehicles (AGV), and have developed into a relatively well-established theoretical system. Since the VSLAM method is increasingly mature, most researchers have prioritized their research into deploying simultaneous localization and mapping (SLAM) into more complex scenarios with longer lifecycles to cope with more challenging issues. For large-scale[4,5] and lifelong[6,7] VSLAM systems, the computational complexity and time cost are too high, resulting in the disjointed information from the real-time input data flow. Therefore, it is crucial to efficiently increase the operating speed of the VSLAM system while guaranteeing its accuracy.

With the development of the compute unified device architecture (CUDA)[8], graphics processors[9,10] are no longer limited to graphics tasks. More researchers have begun to study the use of the graphic processing unit (GPU) for general-purpose computing. Leveraging the computing performance that is tens or even hundreds of times better than the central processing unit (CPU) and combining with the friendly programming environment supported by CUDA architecture, GPU parallel acceleration has been successfully applied in many fields such as data mining, weather prediction, and behavior recognition[11]. Similarly, the establishment of the VSLAM system based on heterogeneous computing[12] has become a hot research content to solve the real-time problem of VSLAM.

For VSLAM development

VSLAM generally includes front-end visual odometry[13], back-end optimization[14] and mapping[15], and loop closing[16]. In 2015, Oriented FAST and Rotated BRIEF SLAM (ORB-SLAM2)[17] was proposed by Mur-Artal and Tardós, which is a classical VSLAM system based entirely on feature points. The program designed a closed-loop detection method based on bag-of-words (BOW) modeling, which can generate sparse 3D maps with centimeter-level accuracy. In contrast to[17], which estimated the camera poses with the Efficient Perspective-n-Point (EPnP) algorithm, ORB-SLAM3[18] employed the pose estimation method based on the PnP algorithm. Although the latter has a certain improvement in accuracy and robustness, its real-time performance is still difficult to guarantee when dealing with large-scale scenarios.

For VSLAM parallel acceleration development

In large-scale scenes, a higher resolution camera with a wide-angle lens is often utilized to obtain wide-view and clearer images. When the image size is too large, the running speed of VSLAM becomes extremely slow due to the large amounts of pixels processed in the feature detection and matching module. Mohammadi and Rezaeian used CUDA to improve the extraction of scale invariant feature sift, which not only ensured the accuracy, but also improved the speed by nearly 30 times through the optimal combination of kernels[19]. Nevertheless, it still cannot be applied in real-time SLAM applications. Parker *et al.* designed a parallel acceleration scheme for feature extraction and feature matching based on the Learned Arrangement of Three Patches (LATCH) descriptors[20]. The scheme was also applied to structure from motion (SFM), achieving nearly ten times faster than the feature description based on Scale-Invariant Feature Transform (SIFT) descriptors while maintaining accuracy. Urban and Hinz extended and improved upon a state-of-the-art SLAM to make it applicable to be rigidly coupled with multi-camera systems by MultiCol model[21]. On this basis, Li *et al.* proposed a CPU-based multi-threading method for parallel image reading, feature extraction and tracking, which solved the load imbalance problem and further improved computational efficiency[22]. Moreover, the feature extraction was implemented via OpenMP in CPU, while the feature matching was implemented in GPU, significantly reducing the computational cost and power consumption compared with ORB-SLAM2. Similar work such as ref[23], by reasonably allocating computing resources between CPU-GPU and adjusting the order of modules in the original feature detection algorithm, improved the execution efficiency of the VSLAM front-end algorithm on a high-performance embedded platform. Nagy *et al.* proposed feature detection based on a lookup table and non-maximal suppression based on cell grids, which further improved the front-end parallel scheme and achieved a new breakthrough in execution speed[24]. The current parallel acceleration schemes for feature detection are limited to separate tasks such as corner detection and non-maximal suppression. In this case, the data needs to be copied multiple times between the video memory and the host, which greatly degrades the overall acceleration effect. For the optimization module involving large-scale matrix solutions, traditional sequential execution algorithms become increasingly incapable when the number of variables increases with runtime in a long-term VSLAM system. The bundle adjustment (BA) scheme, an improvement based on Levenberg-Mardquardt (LM), optimizes the access strategy for Hessian, Schur and Jacobian matrices, with an overall speedup of nearly 30 times. Zheng *et al.* used the preconditioned conjugate gradient (PCG) method and the inexact Newton method to solve the BA problem[25], which improved the computational efficiency by 20 times and reduced memory consumption compared to[26]. However, the method had the problem of low stability. Cao *et al.* applied principal component analysis (PBA) to SFM, only using GPU to implement high-dimensional matrix multiplication, and did not conduct an in-depth study on the parallelization of BA[27]. At present, there are few related researches on the parallel acceleration of the VSLAM back-end optimization and the existing works have the problems of low accuracy, incomplete functions and poor robustness[22].

Current works on the complete (including front-end and back-end) parallel acceleration of SLAM methods are very rare[28]. Lu *et al.*[26] realized a parallelization scheme for the VINS-Mono algorithm[29] by rewriting the flow tracking, nonlinear least-squares optimization, and marginalization program, but its speedup is not significant. To enhance the operation speed of the SLAM system, the paper proposes a relatively complete parallelization scheme of VSLAM method based on heterogeneous computing. For the time-consuming feature extraction and matching at the front end, a full-flow acceleration algorithm is designed to perform the entire process from image input to feature matching on GPU. It mainly includes Gaussian pyramid generation with arbitrary scales, FAST corner extraction based on a lookup table, non-maximal suppression based on grid cell, calculation of feature descriptors, feature matching based on Hamming distance and Random Sample Consensus (RANSAC)-based false matching filtering; for the back-end optimization of VSLAM, the paper uses CUDA to implement the parallel graph optimization based on the Levenberg-Marquardt method. Combined with the characteristics of marginalization in incremental equations and independent observation in graph optimization, the parallelization algorithms including error calculation, construction and update of linear systems, Schur complement reduction and linear equation solving are realized. In each subtask, we make full use of system resources such as shared memory and constant memory, and combine coarse-grained and fine-grained parallelism to optimize the acceleration strategy. Finally, we integrate the above improved front-end and back-end parallel algorithms into a state-of-the-art VSLAM framework, and compare it with other popular methods on the public datasets to verify the effectiveness of the proposed method. The flowchart of the proposed algorithm is shown in Figure 1. The main contributions of the work are listed as follows:

(1) The paper designs a full-flow strategy for feature extraction and matching. After the image is copied from the CPU to the GPU, all computing tasks are independently implemented by the GPU, reducing the time consumption on the multiple data transfers between devices.

(2) The parallel graph optimization method implemented in the paper is a perfect substitute for g2o (general graph optimization), which not only supports basic operations such as adding and removing the vertices and edges, acquiring error, and setting optimization levels on the CPU but also takes full account of the independence of edges and the sparsity of matrices in VSLAM graph optimization. It can be better applied to the global optimization and local optimization of VSLAM.

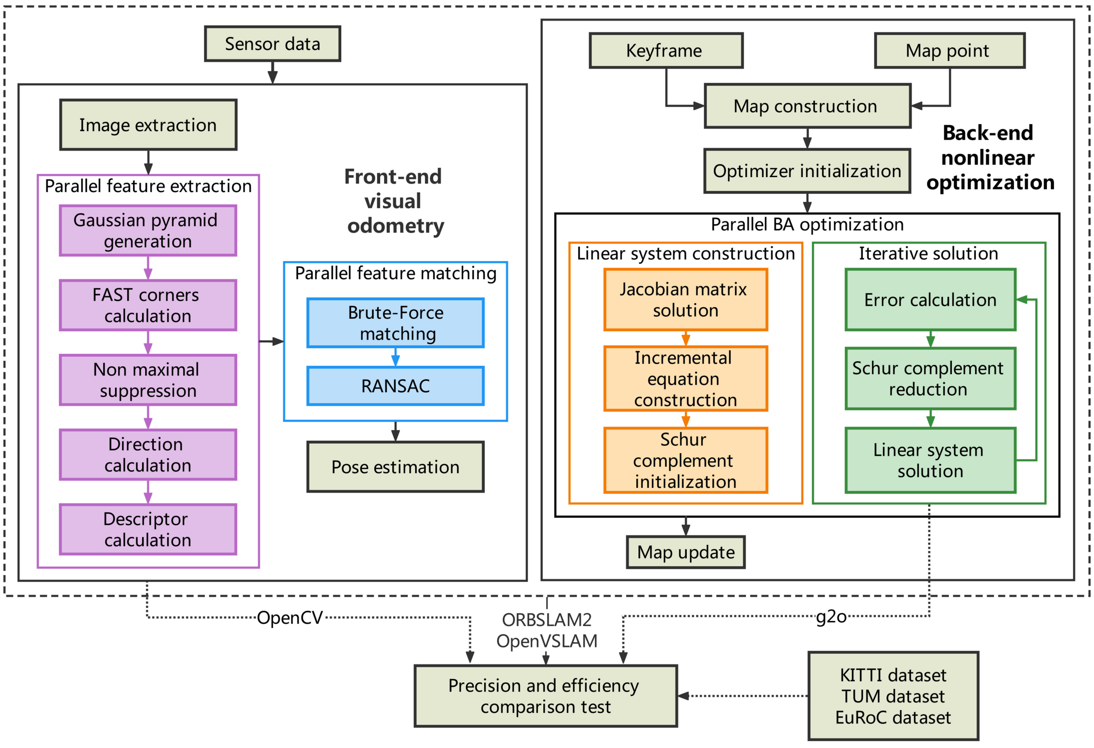

Figure 1. Design scheme of the heterogeneous VSLAM system. VSLAM: Visual simultaneous localization and mapping.

(3) The parallel feature extraction and matching algorithm and parallel graph optimization algorithm proposed in the paper can be called by the VSLAM system in the form of interfaces. The tracking speed of the integrated VSLAM system by parallel modules has been doubled. Moreover, the better performance of the VSLAM system could be achieved by flexibly adjusting the number of variables in practical applications.

2. PARALLEL ACCELERATION FOR VSLAM FRONT-END

Feature extraction and matching is the most basic task in the visual odometry, and is also an extremely time-consuming part due to the multiple submodules involved in feature extraction and the numerous feature points in feature matching. Image feature extraction plays a significant role in detecting a particular type of point in an image and assigning a certain special description to those points. The ORB feature extraction has the properties of high speed and stability and rotation- and scale-invariance, which has become the first choice for feature-based VSLAM methods. Therefore, in the paper, ORB-based feature extraction and matching is selected for the CUDA acceleration.

The pipeline of feature extraction and matching parallel acceleration is shown in Figure 2A. From inputting the image on the CPU side to outputting the result on the GPU side, the system successively performs several sub-tasks such as image pyramid generation, feature point extraction, non-maximal suppression, descriptor calculation and feature matching.

2.1 Image pyramid

An image pyramid is a form of multi-scale representation of the image. During image preprocessing, in order to obtain more scale-invariant feature points, image pyramids need to be constructed on the GPU and saved in the global memory.

The first layer of the image pyramid is derived from the original image input on the CPU. The asynchronous transfer is utilized via the “cudaMemcoy2DAsync()” to copy the data from the host to the device, which saves

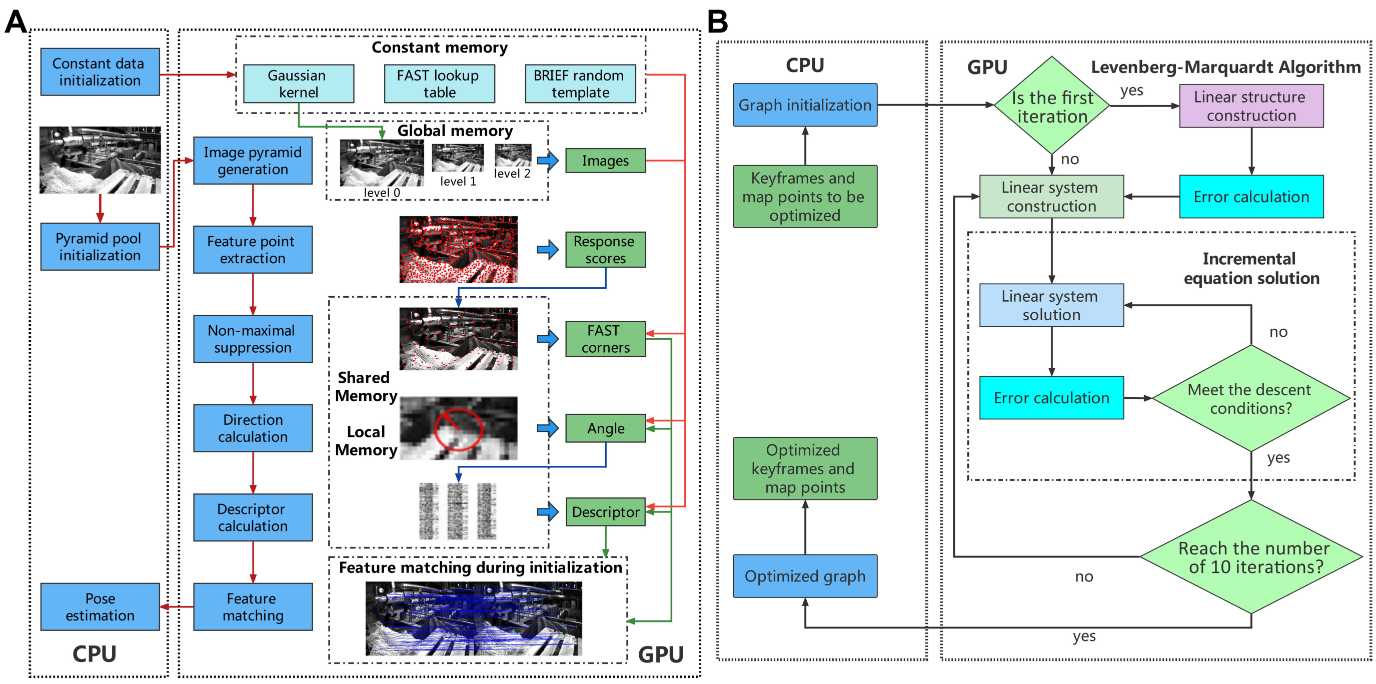

Figure 2. The process of parallel algorithm. (A) The parallel algorithm for feature extraction and matching; (B) The parallel algorithm for optimization. CPU: Central processing unit; GPU: graphic processing unit.

time compared to data transfer through Peripheral Component Interconnect Express (PCI-e) bus. From the second layer, one thread is assigned to process one pixel. For each layer, the image is generated by downsampling the data from the previous layer. To satisfy the diversity of scale factors, the proposed method uses the bilinear interpolation method to calculate the pixel values. In the image preprocessing stage, non-integer scale factors are more likely to generate non-integer pixel coordinates during downsampling than integer scale factors. Therefore, bilinear interpolation is employed to find the four nearest pixel points to that pixel coordinate to calculate the pixel value, which can reduce the visual distortion to some extent. In addition, in order to extract higher quality feature points, the image needs to be initialized with the Gaussian filter before generating the image pyramid corresponding to the pyramid pool initialization of Figure 2A. Image filtering is an identical and independent convolution operation on all pixels of an image, so it is well suited for parallel operations in the GPU. Since the Gaussian template is a fixed two-dimensional array, it can be stored in the constant memory of the CUDA architecture to avoid duplicate memory copies.

2.2 FAST detection

FAST corner is determined by detecting the similarity between the corner candidate p and its sixteen surrounding pixels within a radius of 3 pixels. The label L_x of the pixels around p is taken as one of three states according to

$$L_x = \begin{cases} darker & I_x < I_p - T \\ similar & I_p - T \leq I_x \leq I_p + T \\ brighter & I_x > I_p + T, \end{cases} \tag{1}$$

where T is the detection threshold, and I_p is the pixel value of the corner candidate. If there are continuous N pixels whose labels are "darker" or "brighter", then it is regarded as a corner. Since the corner detection is only related to its own neighborhood, one thread is still assigned to process one pixel for parallel acceleration of detection.

In CUDA, the basic scheduling unit for threads is a wrap. A wrap consists of 32 threads, and execution is most efficient using multiples of 32 as the number of threads per thread block. However, since the number of registers is limited, the registers struggle to meet the computational demands when too many threads are allocated in a thread block, to the point where data is transferred to local memory at a lower rate. In order to improve the utilization of video memory, the allocation of thread blocks is reasonably distributed according to the size of the image to be processed to avoid the waste of resources. The width and height of the target image are represented by "imgcol" and "imgrow", respectively. The thread block and thread lattice are set to two dimensions with a scaling factor of two. The thread-grid allocation strategy is given in:

$$\begin{aligned} dimGrid.x &= \left\lceil \frac{imgcol}{dimBlock.x} \right\rceil \\ dimGrid.y &= \left\lceil \frac{imgrow}{dimBlock.y} \right\rceil \end{aligned} \tag{2}$$

Each thread corresponds to a pixel and the pixel coordinates (x, y) correspond to the threads as follows:

$$\begin{aligned} x &= \text{blockDim}.\ x * \text{blockIdx}.\ x + \text{threadIdx}.\ x \\ y &= \text{blockDim}.\ y * \text{blockIdx}.\ y + \text{threadId}\ x.y \end{aligned} \tag{3}$$

In kernel function, we refer to the look-up table method[24] to avoid the implementation problem of wrap divergence triggered by the if/else statement. The FAST response values are calculated according to the Sum of Absolute Differences (SAD-A) method.

2.3 Non-maximal suppression

In order to avoid duplicate detection and excessive concentration of corners in a certain area, non-maximal suppression is generally used to filter the corners according to their response values. In the paper, each layer of the image pyramid is divided into several rectangular grids using a method similar to that described in ref[24], and the grid size of each layer is adjusted according to the scale factor to ensure the consistency of the grid index among the layers. In the CUDA architecture of NVIDIA, the basic thread scheduling unit, the wrap, contains 32 threads, and the threads in a wrap share the same shared memory. The shared memory has low transfer latency, and all threads in a wrap execute the same instructions. If different threads enter different branches, a wrapdivergence occurs, which has an impact on performance. For non-maximal suppression, the CUDA architecture is well suited to operate in conjunction with wrap. Firstly, the threads in a wrap must exist in the same block; if each grid in the image corresponds to a block, each pixel in the grid is assigned a thread in a wrap unit. In this way, high communication speed can be achieved while avoiding wrap divergence.

After the non-maximal suppression inside the grid, the maximal values in the grids with the same index among different layers are then compared, and the FAST corners with the largest response value are retained and normalized to the original image, so as to reduce the redundant feature points and ensure the uniform distribution of feature points on the image.

2.4 Descriptor calculation

For rotation-invariant features, the "angle" of the corner points needs to be calculated during feature extraction, which correlates to the direction calculation of Figure 2A. The first step is to calculate the grayscale centroid as in:

$$\begin{aligned} m_{10} &= \sum_{-R}^{R}\sum_{-R}^{R} xI(x,y), \\ m_{01} &= \sum_{-R}^{R}\sum_{-R}^{R} yI(x,y), \\ m_{00} &= \sum_{-R}^{R}\sum_{-R}^{R} I(x,y), \end{aligned} \tag{4}$$

where m_{00} is the image moment and the centroid coordinates of the image block are defined as follows:

$$c_x = \frac{m_{10}}{m_{00}}, c_y = \frac{m_{01}}{m_{00}}. \tag{5}$$

Then, the principal direction of the feature point is obtained by connecting the geometric center of the image with the centroid:

$$\theta = \arctan(\frac{c_y}{c_x}) = \arctan(\frac{m_{01}}{m_{10}}). \tag{6}$$

In GPU, the wrap is the basic scheduling unit. All threads within a wrap execute the same instruction at the same time, and share the same block of memory in low transfer latency. Since the direction is generally calculated by selecting a circular area with diameter of 32 pixels, one wrap is used to process one feature point. Therefore, each thread in the wrap is assigned to process one row of pixels, which improves the execution efficiency. During the execution, 32 calculation units are required for the center row, while the number of calculations for the rest of the rows is a fixed value of less than 32. The one-dimensional 32-element array u is constructed to save the number of executions per pixel row, and it is stored in constant memory.

In the image matching, the BRIEF descriptor is generally used to represent and describe the detected feature points with its faster calculation speed and higher matching quality compared with SIFT and Speeded-Up Robust Features (SURF) descriptors. It is determined by 256 random point pairs within the neighborhood window around the feature point, where the random pairs can be obtained by Gaussian sampling.

Since the calculation for feature point descriptors is mutually independent and takes up 32 times as much storage space, 32 threads are allocated for each feature point to improve the running efficiency. In order to speed up the data fetch, "cudaMemcpyToSymbol()" is utilized to save point pairs in the constant memory.

2.5 Feature matching

During initialization, the number of extracted feature points is generally several times more than that of tracking due to the fact that the feature matching needs more accurate map points and initial pose estimation. Hence, the feature matching is very time-consuming. Meanwhile, it is also difficult to guarantee real-time matching even if a local matching method is employed within a window, since there is no such a priori information, such as the known movement change between camera poses. Therefore, the paper implements the parallelization of feature matching by resorting to the OpenCV CUDA module, which mainly includes searching for point pairs based on Hamming distance and culling false matchings based on RANSAC algorithm.

For coarse feature matching, the BruteForce algorithm based on the Hamming distance is applied by calculating the set with the minimum distance between two sets of feature vector sets as a matching point pair. Since the size of the feature vectors is fixed, it will be set as $BlockSize$. When performing the initial feature matching, $dimBlock.x$ = dimBlock $.y = BlockSize$. The dimension of the thread grid is set to one dimension, and the parameters are given in

$$dimGrid.x = \left\lceil \frac{n}{BlockSize} \right\rceil \tag{7}$$

In the matching process, n threads are firstly allocated, with thread id as $queryIdx = BlockIdx.x * BlockSize + ThreadIdx.y$, to calculate the Hamming distance of each minimum. The Hamming distance is defined as the number of different positions corresponding to two sets of binary codes with the same length. In the actual calculation process, the distance needs to be compared $BlockSize$ times. Therefore, the thread id is set to $ThreadIdx.x$, and the total number of threads in the program is $n * BlockSize$. Thereby after initial matching, the coarse matching point pairs are obtained.

For fine feature matching, the RANSAC algorithm parallelizes the scoring of coarse match pairs and the removal of false matches. The iterative process is executed in parallel, followed by the matching pair evaluation process. The computations of the model with random sampling are independent, as is the evaluation for each matched pair; hence, both steps can be accelerated by CUDA parallel computation. If the number of randomly sampled point set groups is m, the number of matched point pairs is n. $\text{dimBlock}.x = \min\{512, \text{n}\}$ is set, and the thread block parameters are as given in

$$\text{dimGrid}.x = \left\lceil \frac{m * n}{\text{dimBlock}.x} \right\rceil \tag{8}$$

3. PARALLEL ACCELERATION FOR VSLAM BACK-END

The nonlinear optimization of VSLAM will eventually be transformed into solving a sparse matrix problem, and the scale of the matrix depends on the number of variables which needs to be optimized. Although the complexity of solution can be reduced by marginalization, it is still difficult for BA to run in real-time when the scale of variables to be optimized is too large. Especially, when SLAM detects the loop-closure and carries out the global optimization, it is necessary to adjust the poses of all keyframes and relevant map points. Therefore, the paper proposes a BA scheme based on g2o[30] by introducing GPU parallel computing and improving the traditional BA process to adapt to parallel design, which mainly includes the reconstruction of key and time-consuming parts such as error calculation and linear solution in g2o, memory scheduling optimization and task allocation. Therefore, the running speed can be greatly improved while the accuracy remains unchanged. The specific process can be seen in Figure 2B.

3.1 Construction for the linear structure

Creating the linear structure is the process of building incremental equations. Its main purpose is to build the structures of all matrices used in subsequent solutions. Therefore, it is necessary to allocate fixed memory to the relevant data structure in the GPU.

Here, take the example of the creation of coefficient H. H_{pp} and H_{ll} are diagonal block matrices, so the space of the graphics memory to be allocated is determined by the number, size and data type of the non-zero blocks on their own diagonals. The number of non-zero blocks in H_{pp} is the same as the number of pose vertices denoted by poses_size. Since the non-zero block is a 6*6 matrix, if a double precision floating-point type is used to save data, the space of the sizeof(double)*36*poses_size needs to be allocated. For H_{pl}, in addition to allocating graphics memory according to non-zero blocks, it is also necessary to save the number and position of non-zero blocks and the index of corresponding edges for subsequent Schur complement reduction. According to the physical meaning of H_{pl}, when the j-th map point can be observed from the i-th camera, H_{pl} has a non-zero block at position (i, j). Since edges do not affect each other, we assign corresponding threads to calculate the number of non-zero blocks in parallel depending on the number of edges, and the index information could be obtained according to the prestored H_{pl} non-zero block set in the device.

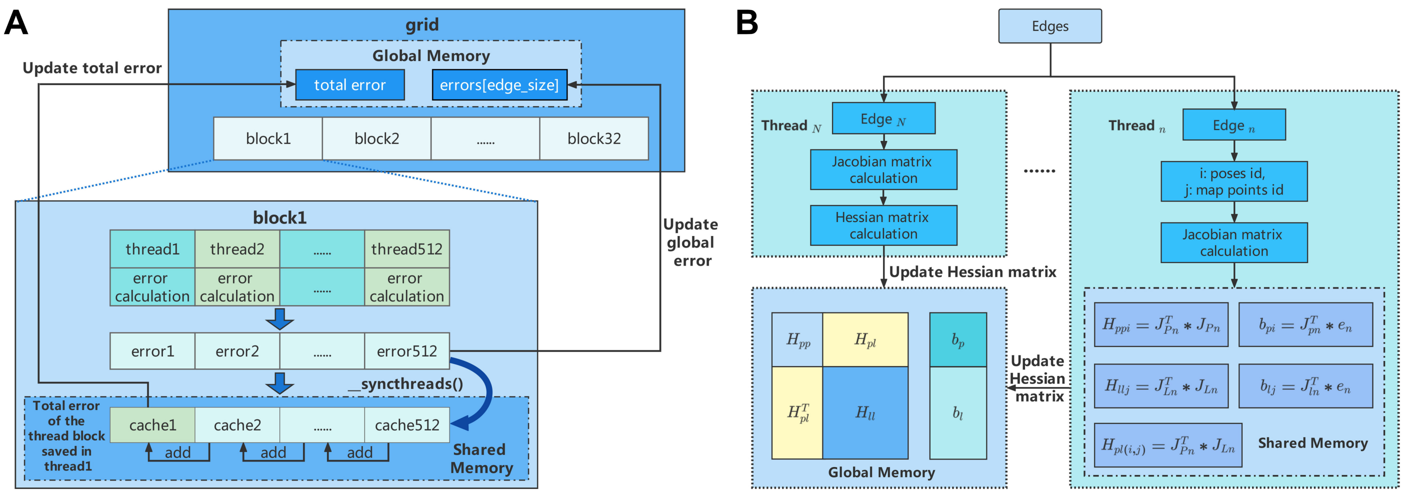

Figure 3. The process of error calculation and the construction for the linear system. (A) The chart of the parallel error calculation; (B) The flow chart of parallel solution for linear system update.

3.2 Error calculation

The error calculation of each edge in BA only involves the pose of the camera and the 3D coordinates of its connected map points, and independent threads can be used to realize the parallel calculation of the errors to improve the calculation speed. Nevertheless, unlike the previous calculation, the errors calculated here need to be accumulated, indicating that each thread is not completely independent, and the data needs to be interactive. However, the more thread blocks are called, the more complex the corresponding resources are allocated. As a result, assigning the appropriate number of threads is significant. Considering the shared memory can be used within thread blocks, we divide the sum operation into two steps. The first step is to add errors in all thread blocks and save them in shared memory; the second step is to sum the errors in the shared memory. The steps above largely avoid the data interaction among thread blocks and greatly improve the running efficiency. The overall process of parallel error calculation is given in Figure 3A.

3.3 Construction for the linear system

Building a linear system is the most critical step in the optimization process and the basis of the subsequent solution for the incremental equation. The main task of this part is to solve the Jacobian matrix of the error about the poses and map points according to the data obtained in the first two parts, and then determine the specific forms of H_{pp}, H_{pl}, H_{ll}, b_p and b_l.

First, the Jacobian matrix corresponding to each edge needs to be calculated. There are two types of Jacobian matrix: J_p and J_l. The calculation of each Jacobian matrix is a completely independent operation. Therefore, the number of threads allocated equals that of edges. For the n-th thread, which corresponds to the n-th edge, the id of the pose vertex and the map point vertex corresponding to this edge are i and j, respectively. When the thread is executed, the i-th non-zero block of H_{pp} will be superposed once $J_{pn}^T * J_{pn}$. Other matrices in the incremental equation operate similarly. The process of updating the linear system is shown in Figure 3B.

3.4 Solution for the linear system

The linear system solution is divided into two parts: Schur complement reduction and linear equation solution. The matrices to be maintained for marginalization are:

$$H_{Sc} = H_{pp} - H_{pl} * H_{ll}^{-1} * H_{pl}^T, \tag{9}$$

$$b_{Sc} = -b_p + H_{pl} * H_{ll}^{-1} * b_l. \tag{10}$$

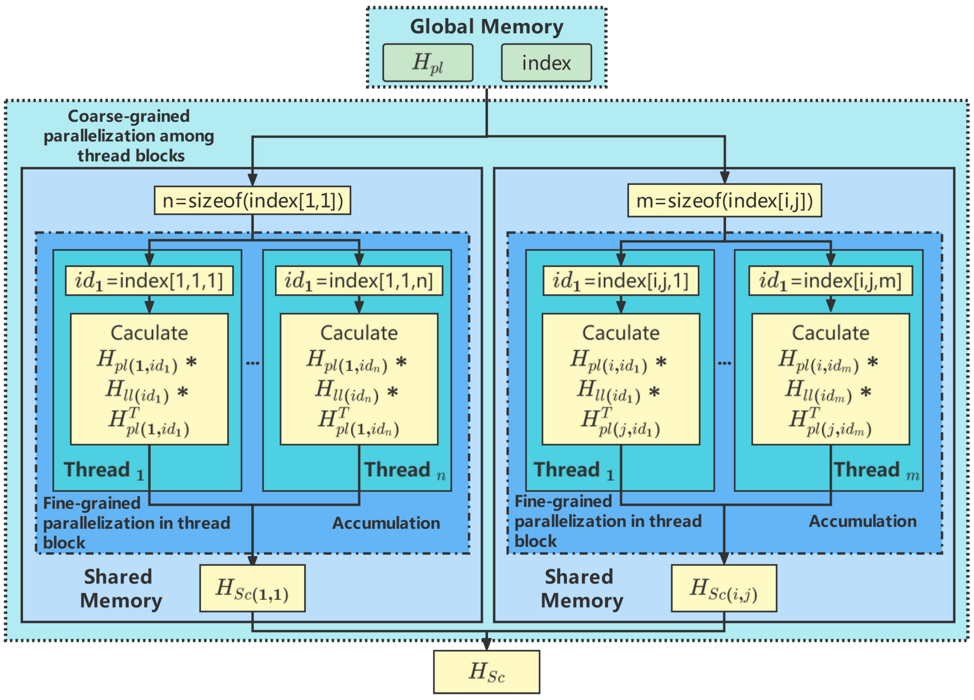
Global Memory
H_{pl}
index
Coarse-grained parallelization among thread blocks
n=sizeof(index[1,1])
m=sizeof(index[i,j])
id_1=index[1,1,1]
id_1=index[1,1,n]
id_1=index[i,j,1]
id_1=index[i,j,m]
Caculate $H_{pl(1,id_1)}$ * $H_{ll(id_1)}$ * $H^T_{pl(1,id_1)}$
Caculate $H_{pl(1,id_n)}$ * $H_{ll(id_n)}$ * $H^T_{pl(1,id_n)}$
Caculate $H_{pl(i,id_1)}$ * $H_{ll(id_1)}$ * $H^T_{pl(j,id_1)}$
Caculate $H_{pl(i,id_m)}$ * $H_{ll(id_m)}$ * $H^T_{pl(j,id_m)}$
Thread $_1$
Thread $_n$
Thread $_m$
Fine-grained parallelization in thread block
Accumulation
Shared Memory
$H_{Sc(1,1)}$
$H_{Sc(i,j)}$
H_{Sc}

Figure 4. Flow chart of parallel solution for H_{Sc}.

It can be seen from the formulas that H_{Sc} and b_{Sc} are obtained by summing two matrices, and parallel computing with CUDA is very suitable for matrix superposition.

For the example of the coefficient H_{Sc}, we copy H_{pp} to the corresponding position in H_{Sc} as its initialized structure, and take the number of non-zero blocks of H_{pp} as the number of threads, and each thread performs a copy operation. Assuming that the number of camera poses is m and the number of map points is n, the H_{Sc} can be divided into m*m sub-blocks. The sub-block in the position (i, j) is:

$$H_{Sc(i,j)} = \sum_{x=1}^{n} H_{pl(i,x)} * H^{-1}_{ll(x)} * H^T_{pl(x,j)}, \tag{11}$$

where $H_{pl(i,j)}$ is the i-th row and j-th column of sub-block H_{pl}, and $H_{ll(x)}$ is the x-th non-zero of sub-block H_{ll}. That is, each sub-block of H_{Sc} needs to be added n times. Due to the sparse structure of H_{pl}, the number of calculations will be much less than n. The calculation number of $H_{Sc(i,j)}$ is related to the i-th and j-th rows of H_{pl}. If there are ν identical column indexes of the non-zero block in row i and row j of H_{pl}, there will be ν sum operations. During each iteration, the index of H_{pl} does not change, so all non-zero block indexes can be pre-calculated in the part of the linear structure construction. Since the calculation of each sub-block in H_{Sc} is independent, and the calculation of each sum-term in the sub-block is also independent, H_{Sc} is computed using a two-level parallel mode, with coarse-grained parallelization among non-zero blocks and fine-grained parallelization within non-zero blocks, further improving the operation efficiency. The calculation of b_{Sc} also adopts the same strategy. Figure 4 is the flow chart for solving H_{Sc}.

In order to make full use of the limited storage resources of the graphics processor, Compressed Sparse Row (CSR) format is adopted for sparse matrix H_{Sc} to store data. Compared with common matrix storage, CSR format can avoid the waste of space. Finally, the paper uses the linear solver cuSolver provided by CUDA to solve the incremental equation.

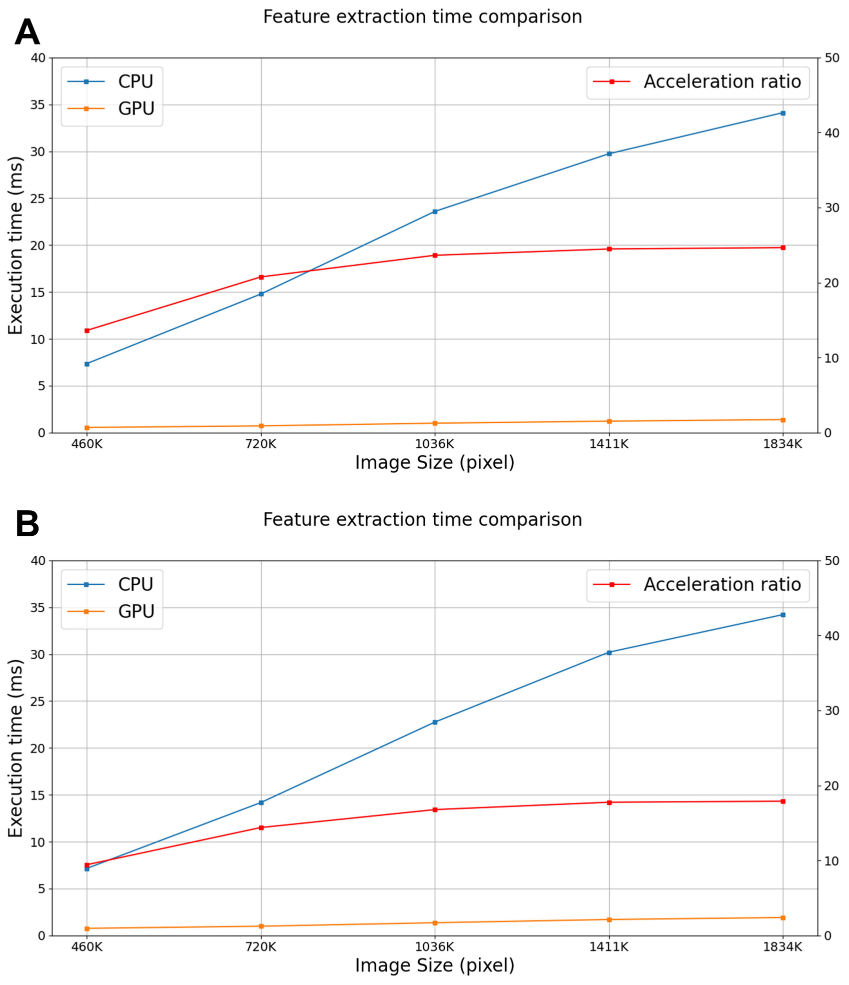

Figure 5. The runtime comparison of feature extraction with different graphics card models. (A) RTX 2080Ti; (B) GTX 1050Ti.

4. EXPERIMENT

In this section, we evaluate our proposed CUDA parallel acceleration method for the feature detection and matching module in front-end and the optimization module in back-end. Meanwhile, the optimized accelerated modules are integrated into a common VSLAM pipeline, referred to as CUDA-SLAM. The proposed method is compared with the state-of-the-art VSLAM methods with multiple public datasets. The experiment is implemented on a workstation with AMD Ryzen Threadripper 1950X 16-Core CPU and three different NVIDIA graphics cards whose properties are summarized in Table 1.

4.1 Feature extraction and matching

The feature extraction algorithm based on CUDA is a complete acceleration scheme for ORB feature extraction. It realizes the functions of pyramid generation, feature point extraction, non-maximal suppression, direction calculation and descriptor calculation.

To verify the performance of the proposed method, we select images with pixels of 460 K, 720 K, 1,036 K, 1,411 K and 1,834 K, respectively, to measure the runtime of the serial feature extraction algorithm based on OpenCV-CPU and the parallel feature extraction algorithm based on CUDA acceleration. Figure 5 displays the comparison of the feature extraction time with different graphics card models.

It can be seen from Figure 5 that the parallel feature extraction algorithm achieves superior execution times. When extracting features from images with 1834K pixels in RTX 2080Ti, the speedup can reach 19.7, demon-

Table 1. Image processor parameters

Property	NVIDIA GeForce RTX 2080Ti	NVIDIA GeForce RTX 1050Ti	NVIDIA Jetson AGX Orin
Number of CUDA cores	4,352	768	2,048
Video memory capacity	11 GB	4 GB	32 G
Video memory bandwidth	616 GB/s	112 GB/s	204.8 GB/s
Video memory bus width	352bit	128bit	256bit
Video memory type	GDDR6	GDDR5	LPDDR5
Core architecture	Turing	Pascal	Ampere

CUDA: Compute unified device architecture.

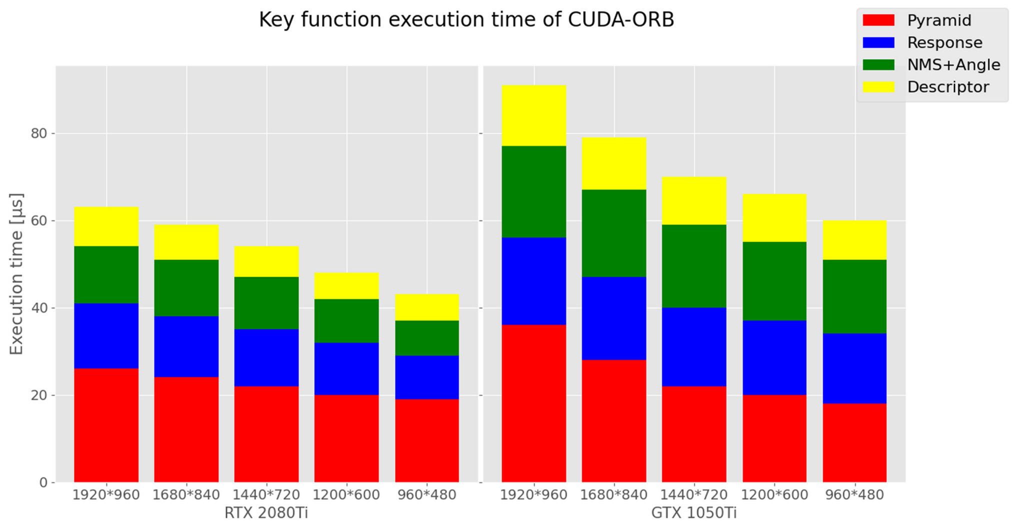

Figure 6. The runtime comparison of key functions in parallel feature extraction.

strating that the CUDA parallel feature extraction algorithm proposed in the paper is effective. It is also shown that the acceleration effect of the two types of graphics cards is significantly different due to the difference between the number of CUDA cores and the difference between the bandwidth of graphics memory. At the same time, the speedup increases with the growth of the image size, but the acceleration rate slows down with the increase of the image size, which conforms to the heterogeneous acceleration characteristics. For the improved acceleration algorithm, the whole image needs to be copied to the graphics card device first. The higher the image resolution, the more data will be copied to the graphics card device, which consumes a lot of time and is counted in the pyramid part. In order to avoid chance errors, the images of different pixels are executed 50 times separately and the average execution time is taken as the statistical result. The execution time of each key function in the feature extraction can be seen in Figure 6. As can be seen in Figure 6, pyramid generation consumes the most time in the front-end, and the overall time consumed decreases significantly as the performance of the graphics card increases.

The performance of feature matching is mainly evaluated by two indicators:

(1) Accuracy: this indicator refers to the number of successful matching point pairs whose Hamming distance is less than a certain threshold.
(2) Running speed: this indicator refers to the average runtime in the process of matching a certain number of feature points.

Figure 7 shows the results of the proposed parallel matching algorithm. It can be seen that the successful matching rate is high. In order to further verify the accuracy of the algorithm, we calculate the matching

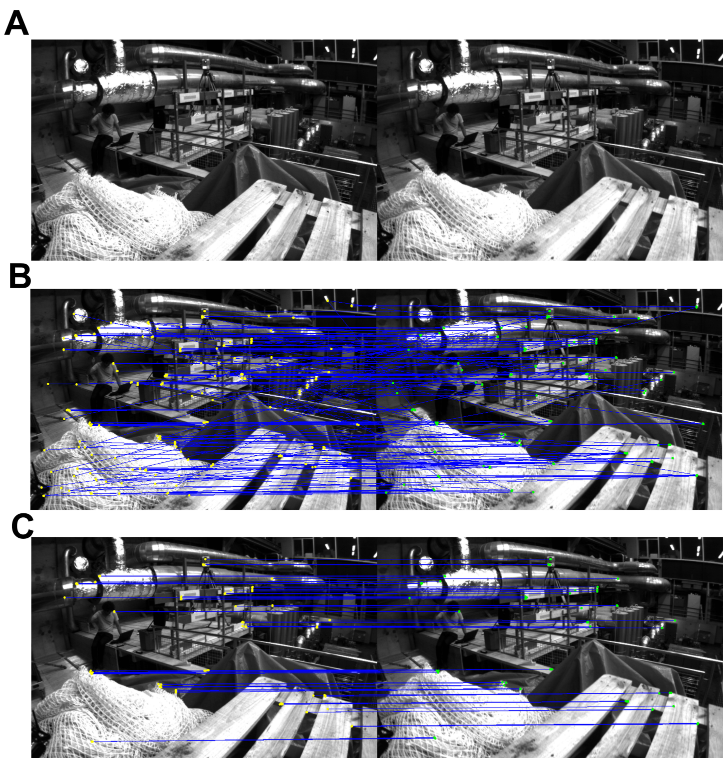

Figure 7. The feature matching results of different algorithms. (A) Original images in EuRoC sequence MH 01; (B) Matching result based on Brute-Force algorithm; (C) Matching result based on RANSAC algorithm. RANSAC: Random Sample Consensus.

Table 2. Accuracy comparison of feature matching

	Serial matching algorithm		Parallel matching algorithm	
Number of feature points	Number of correct matches	Successful matching rate	Number of correct matches	Successful matching rate
100	99	99%	98	98%
200	194	97%	198	99%
400	389	97.25%	395	98.75%
800	796	99.5%	792	99%
1,600	1,590	99.38%	1,586	99.13%
3,200	3,184	99.5%	3,189	99.66%

accuracy for different numbers of feature point pairs, and compare it with the feature matching algorithm implemented by OpenCV-CPU. The results are shown in Table 2. It can be seen that the matching accuracy of the two algorithms is basically identical, indicating that the acceleration algorithm proposed in the paper is effective.

Time consuming test is implemented on two 960*480 pixels images that have certain overlapping scenes. We limit the number of features extracted in each image, and calculate the average runtime of serial and parallel matching algorithms after executing 50 times. The results are shown in Figure 8.

As can be seen from Figure 8, the parallel feature matching algorithm takes much less time than the serial matching algorithm. With the increase of the number of feature points, the speedup on both graphics cards has improved, indicating that GPU has a better acceleration effect on large-scale data, but the improvement of acceleration ratio slows down with the increase of the number of feature points. This is because data transfer

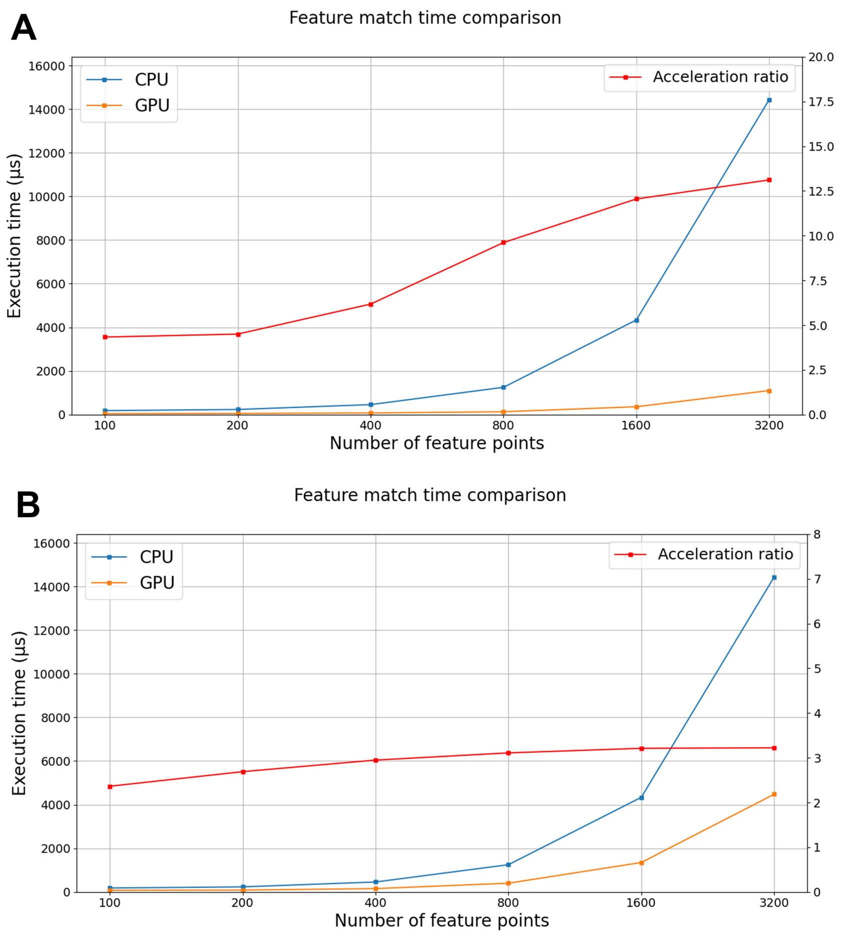

Figure 8. The runtime comparison of feature extraction with different graphics card models. (A) RTX 2080Ti; (B) GTX 1050Ti. CPU: Central processing unit; GPU: graphic processing unit.

between GPU and CPU takes a lot of time, which indirectly shows that the time of parallel feature matching algorithm is mainly spent on data transmission. By comparing the runtime of the two graphics cards, it can be seen that the data transmission efficiency and computing performance of GTX 1050Ti are far lower than that of RTX 2080Ti.

Through comparisons on accuracy and runtime, it can be seen that the parallel feature matching algorithm based on CUDA proposed in the paper is roughly the same as the serial OpenCV algorithm in terms of matching accuracy, but has significantly improved in terms of running efficiency.

4.2 Bundle adjustment

In this section, we compare the parallel optimization algorithm proposed in the paper with the g2o algorithm running on CPU using different linear solvers, such as Eigen, Csparse and PCG. The test data for the experiments is obtained from the map generated by ORB-SLAM running on Kitti 00 sequence and saved in the form of a graph which includes keyframe-pose vertices, map point vertices, corresponding binary edges and ternary edges. The data are categorized as different scales, and the runtime for ten iterations in optimization can be seen in Table 3.

To analyze the results more intuitively, we draw the runtime curves of the two algorithms for different data sizes, as shown in Figure 9A, where the horizontal axis represents the number of poses and the vertical axis represents the optimization time with ten iterations. As seen in Figure 9A, the parallel algorithm runs faster

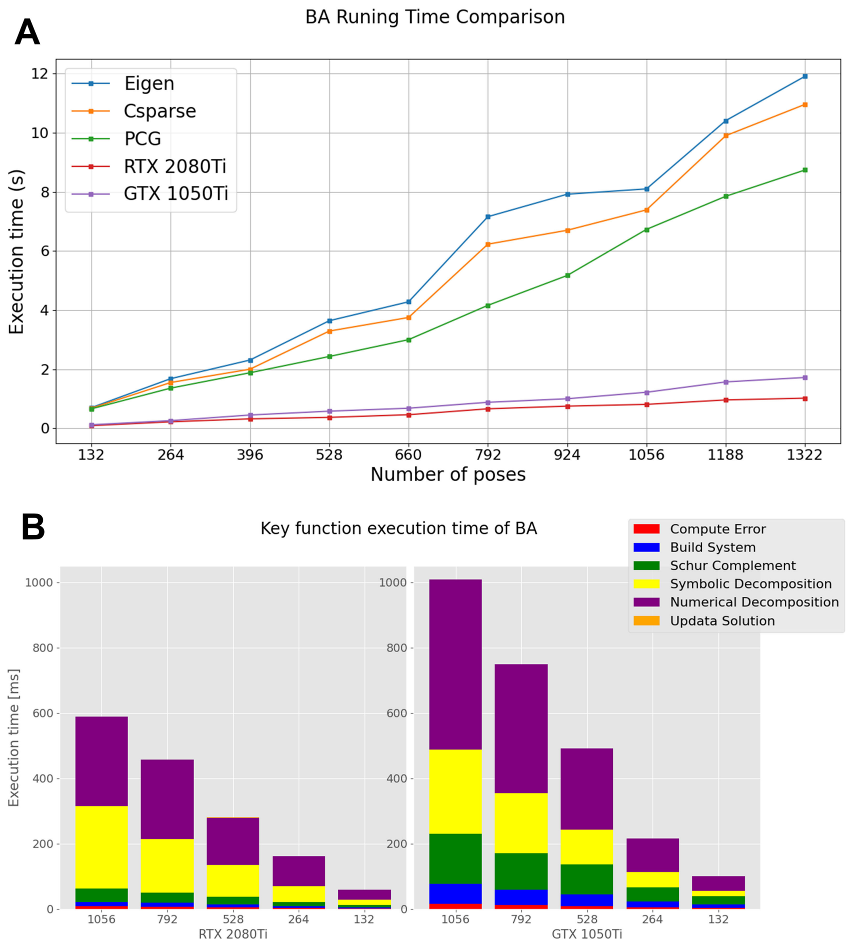

Figure 9. The runtime comparison of different methods. (A) Runtime comparison of bundle adjustment; (B) Runtime comparison of key functions in parallel bundle adjustment.

Table 3. Runtime comparison of bundle adjustment

Number of poses	Map points	Number of edges	Parallel time consuming(s)		Serial time consuming(s)		
			2080Ti	1050Ti	Eigen	Csparse	PCG
132	17,333	64,201	0.09	0.12	0.70	0.68	0.66
264	34,968	135,702	0.22	0.26	1.68	1.55	1.36
396	48,700	185,364	0.32	0.45	2.31	2.00	1.88
528	61,442	239,182	0.37	0.58	3.64	3.29	2.43
660	75,203	289,890	0.46	0.68	4.28	3.75	3.00
792	88,736	348,542	0.66	0.88	7.16	6.23	4.16
924	103,255	402,374	0.75	1.00	7.92	6.70	5.17
1056	109,978	458,623	0.81	1.22	8.10	7.39	6.73
1188	120,817	511,519	0.96	1.57	10.41	9.90	7.85
1322	133,383	561,116	1.02	1.72	11.91	10.96	8.74

than the serial algorithm for different data sizes, and the speedup is more obvious as the number of keyframes increases. Figure 9B shows the runtime of the key functions in BA algorithm for different numbers of poses.

Since our approach focuses on parallelization without changing mathematics and tactics in g2o, we compare the accuracy of the proposed parallel optimization algorithm with the Levenberg-Marquardt algorithm implemented in g2o, and the test results for different data sizes are shown in Table 4. Through two comparisons in Tables 3 and 4, it can be seen that the proposed parallel optimization scheme has a 5-12 times speedup, while maintaining the same accuracy as g2o.

Table 4. Accuracy comparison of bundle adjustment

Data id	Number of edges	Original error	Parallel algorithm error (pixel)		Serial algorithm error(pixel)	
			1 iteration	10 iterations	1 iteration	10 iterations
0	64,201	0.588	0.549	0.500	0.549	0.500
1	135,702	0.579	0.549	0.502	0.549	0.502
2	185,364	0.562	0.538	0.500	0.538	0.500
3	239,182	0.563	0.542	0.511	0.542	0.511
4	289,890	0.547	0.530	0.503	0.530	0.503
5	348,542	0.555	0.530	0.503	0.530	0.503
6	402,374	0.544	0.523	0.500	0.523	0.500
7	458,623	0.569	0.543	0.522	0.543	0.522
8	511,519	0.575	0.551	0.523	0.551	0.523
9	561,116	0.629	0.600	0.570	0.600	0.570

Table 5. Thread parameters for different tasks

Section	BlockDim.x	BlockDim.y	GridDim.x	GridDim.y
Image pyramid	$\min(64, \frac{imgcol+32-1}{32} \times 32)$	2	$\frac{imgcol+BlockDim.x-1}{BlockDim.x}$	$\frac{imgrow+BlockDim.y-1}{BlockDim.y}$
Fast corner detection	32	4	$\frac{imgcol+BlockDim.x-1}{BlockDim.x}$	$\frac{imgrow+BlockDim.y-1}{BlockDim.y}$
Non-maximal suppression	$cellcol_l$	$\max(1, \min[\frac{128}{cellcol_l}, cellrow_l])$	$\frac{imgcol_l}{cellcol_l}$	$\frac{imgrow_l}{cellrow_l}$
Direction calculation	32	8	1	$\frac{Npoints}{BlockDim.y}$
Descriptor calculation	32	8	1	$\frac{Npoints}{BlockDim.y}$
Creating linear structures	32	32	$\frac{Nedges}{32*32}$	1
Error calculations	32	16	32	1
Building linear systems	512	1	$\frac{Nedges}{512}$	1
Solving linear systems	512	1	$\frac{Ncalculations}{BlockDim.x}$	1
Note	(1) BlockDim and GridDim is the size of the thread block and thread grid (2) $imgrow_l$ and $imgcow_l$ is the size of the image, where l represents the level of the current image pyramid (3) $cellrow_l$ and $cellcow_l$ is the size of the grid used in non-maximal suppression (4) $Nedges$ is the number of the edges in BA (5) $Ncalculations$ is the number of the calculation to the H_{Sc} or b_{Sc}			

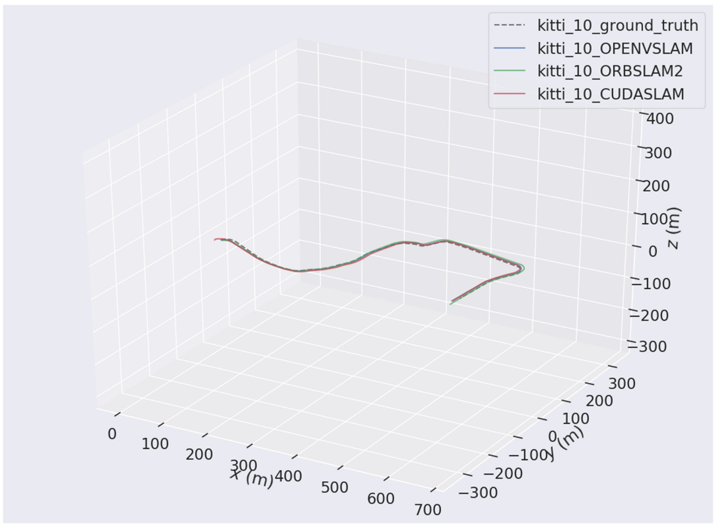

Figure 10. Tracking results for different VSLAM systems. VSLAM: Visual simultaneous localization and mapping.

Table 6. Tracking accuracy comparison of different VSLAM systems

	Evaluation index	ORB SLAM2	Open VSLAM	CUDA-SLAM
RPE (m)	Max	0.027	0.057	0.036
	Mean	0.008	0.008	0.012
	Median	0.006	0.004	0.010
	Rmse	0.010	0.013	0.014
	Std	0.006	0.009	0.007
APE (m)	Max	0.991	2.344	1.474
	Mean	0.395	0.511	0.645
	Median	0.361	0.402	0.014
	Rmse	0.448	0.684	0.720
	Std	0.213	0.455	0.320

VSLAM: Visual simultaneous localization and mapping; RPE: relative pose error; APE: absolute pose error; SLAM: simultaneous localization and mapping.

Table 7. Runtime comparison of key modules in different VSLAM systems

	ORB-SLAM2	OpenVSLAM	CUDA-SLAM
Front-end	48 ms	52 ms	12 ms
Back-end	138 ms	144 ms	41 ms

VSLAM: Visual simultaneous localization and mapping; CUDA: compute unified device architecture; SLAM: simultaneous localization and mapping.

4.3 Heterogeneous VSLAM system

It is shown that the parallel feature extraction and matching algorithm in Section III-A and the BA algorithm in Section III-B based on CUDA have a significant improvement in running efficiency with approximately the same accuracy as the serial algorithm. Therefore, we integrate the above-mentioned modules with our proposed accelerated nodes to realize the real-time performance of the common VSLAM system, and the detailed thread parameters are shown in Table 5. The improved is called CUDA-SLAM and is compared with the popular SLAM methods including ORB-SLAM[17,18] and OpenVSLAM[31]. The data used to evaluate the errors is from KITTI sequence 10, and the absolute pose error (APE) and relative pose error (RPE) are used to evaluate the accuracy of VSLAM methods. The main parameters are used as follows: the number of feature points is set to 2000, and the covisible keyframe threshold is set to 15; that is, an edge between two keyframes exists if they share observations of the same map points (at least 15).

The tracking results are shown in Figure 10, and the RPE and APE for the VSLAM systems are presented in Table 6. Qualitative and quantitative experiments verify that the error of the proposed scheme under each metric is basically the same as that of OpenVSLAM, indicating that the CUDA-SLAM proposed in the paper meets the accuracy requirements of the general VSLAM model.

Table 7 records the average runtime of the SLAM front-end and back-end. According to Table 7, the efficiency of each module is significantly improved, proving the feasibility of heterogeneous VSLAM systems consisting of the parallel acceleration modules proposed in the paper. Combined with the previous results of accuracy tests, the application of CUDA acceleration to VSLAM key modules can greatly improve the running speed without loss of accuracy, which is very helpful to realize the real-time large-scale VSLAM systems.

The stability and accuracy of SLAM system is directly related to the scale of data. To a certain extent, increasing the number of feature points and keyframes for local optimization can improve the accuracy of tracking and mapping. Therefore, we expand the data scale by doubling the number of feature points and increasing the

Table 8. Accuracy and runtime comparison between OpenVSLAM and CUDA-SLAM

Project	Open VSLAM	CUDA-SLAM1	CUDA-SLAM2	CUDA-SLAM3
Keyframe increase scale	0%	0%	50%	50%
Number of feature points	2,000	4,000	2,000	4,000
RMSE (APE)	0.684 m	0.611 m	**0.550 m**	0.327 m
RMSE (RPE)	0.013 m	0.011 m	0.010 m	**0.008 m**
Tracking time	52 ms	16 ms	**12 ms**	16 ms
Optimization time	144 ms	**41 ms**	53 ms	53 ms

The data with the best performance is bolded in the table. VSLAM: Visual simultaneous localization and mapping; CUDA: compute unified device architecture; SLAM: simultaneous localization and mapping; RMSE: root mean square error; RPE: relative pose error; APE: absolute pose error.

Table 9. Runtime comparison of the tracking module

	ORB-SLAM2		OpenVSLAM		CUDA-SLAM	
Sequence	Median	Mean	Median	Mean	Median	Mean
T1	0.0288 s	0.0291 s	0.0451 s	0.0469 s	0.0124 s	0.0191 s
T2	0.0296 s	0.0298 s	0.0527 s	0.0498 s	0.0119 s	0.0192 s
T3	0.0294 s	0.0295 s	0.0457 s	0.0467 s	0.0118 s	0.0186 s
T4	0.0295 s	0.0297 s	0.0563 s	0.0575 s	0.0125 s	0.0198 s
T5	0.0290 s	0.0291 s	0.0543 s	0.0547 s	0.0126 s	0.0228 s
T6	0.0288 s	0.0289 s	0.0520 s	0.0509 s	0.0134 s	0.0219 s
Average	0.0292 s	0.0293 s	0.0507 s	0.0512 s	0.0124 s	0.0202 s

SLAM: Simultaneous localization and mapping; VSLAM: Visual simultaneous localization and mapping; CUDA: compute unified device architecture.

number of keyframes by 50%, then compare the tracking time and trajectory accuracy of OpenVSLAM and CUDA-SLAM. The results are shown in Table 8.

It can be shown in Table 8 that CUDA-SLAM has significantly improved the tracking accuracy after expanding the data scale, but still consumes much less time than the OpenVSLAM algorithm. It is further verified that the parallel SLAM algorithm in the paper has significant advantages in terms of accuracy and runtime. Therefore, in practical applications, the best performance of VSLAM system can be achieved by flexibly adjusting the number of feature points and keyframes.

To better meet the practical application circumstances, an embedded GPU named Jetson AGX Orin is introduced. Further experiments are conducted on the above hardware platform with TUM datasets to verify the effectiveness of the parallel tracking module more comprehensively. The runtime comparison of the tracking module is shown in Table 9. The performance of the proposed tracking algorithm is improved by 31% compared with the traditional algorithm under the same test environment. It can be seen that our algorithm outperforms traditional algorithms under a variety of open datasets and is stable under the embedded GPU hardware.

5. CONCLUSION

In the paper, we propose a parallel scheme on the key modules of VSLAM system based on CUDA for the large-scale computational tasks with high complexity in the tracking and optimization of VSLAM system. By improving the modules of feature extraction and matching as well as BA, the parallelization of the algorithm is achieved on the GPU. Compared with the traditional sequential execution methods, the speedups of feature extraction, feature matching and BA are respectively 10-20, 5-13 and 10 times, while maintaining accuracy. At last, the proposed front-end and back-end parallel algorithms are migrated to OpenVSLAM. The results show that the tracking accuracy of CUAD-SLAM is basically identical to the state-of-the-art methods with the same settings but the running speed is significantly improved.

Furthermore, when the data scale increases exponentially, the runtime of CUDA-SLAM is still much lower than that of traditional methods. Enhancing real-time capabilities is crucial because real-time responsiveness is a key requirement in many real-world applications. In VSLAM system, loop closure detection is also an important part. It requires determining whether the robot has reached the previous position by comparing the current frame with the reference keyframe based on BOW model, where the similarity calculations are repetitive and independent operations that are feasible to accelerate. Therefore, the parallelization of loop closure detection is considered to be implemented to improve the performance of VSLAM system in the future.

DECLARATIONS

Acknowledgments

We are grateful for the efforts of our colleagues in the Sino-German Center of Intelligent Systems.

Authors' contributions

Made substantial contributions to the research process and wrote the original draft: Shu Z, Liu Y, Hou C
Performed data acquisition: Xu S, Lv T
Provided guidance and support: Liu H, Dong Y

Availability of data and materials

Not applicable.

Financial support and sponsorship

This work was supported by the National Natural Science Foundation of China under Grant 61873189 and Grant 62088101, the Shanghai Municipal Science and Technology Major Project under Grant 2021SHZDZX0100, and the 19th Experimental Teaching Reform Fund of Tongji University under Grant 0800104314.

Conflicts of interest

All authors declared that there are no conflicts of interest.

Ethical approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Copyright

REFERENCES

1. Sharafutdinov D, Griguletskii M, Kopanev P, et al. Comparison of modern open-source visual SLAM approaches. *J Intell Robot Syst* 2023;107:43. DOI
2. Cai D, Li R, Hu Z, Lu J, Li S, Zhao Y. A comprehensive overview of core modules in visual SLAM framework. *Neurocomputing* 2024:127760. DOI
3. Deng T, Chen Y, Zhang L, et al. Compact 3d gaussian splatting for dense visual slam. arXiv. [Preprint.] Mar 17, 2024 [accessed 2024 Aug 27]. Available from: https://doi.org/10.48550/arXiv.2403.11247.
4. Matsuki H, Tateno K, Niemeyer M, Tombari F. Newton: neural view-centric mapping for on-the-fly large-scale slam. *IEEE Robot Autom Lett* 2024;9:3704-11. DOI
5. Yang K, Cheng Y, Chen Z, Wang J. SLAM meets NeRF: a survey of implicit SLAM methods. *World Electr Veh J* 2024;15:85. DOI
6. Chen B, Zhong X, Xie H, et al. SLAM-RAMU: 3D LiDAR-IMU lifelong SLAM with relocalization and autonomous map updating for accurate and reliable navigation. *Ind Robot* 2024;51:219-35. DOI

7. Leonardi M, Stahl A, Brekke EF, Ludvigsen M. UVS: underwater visual SLAM-a robust monocular visual SLAM system for lifelong underwater operations. *Auton Robots* 2023;47:1367-85. DOI
8. Al-Mouhamed MA, Khan AH, Mohammad N. A review of CUDA optimization techniques and tools for structured grid computing. *Computing* 2020;102:977-1003. DOI
9. Ortiz J, Pupilli M, Leutenegger S, Davison AJ. Bundle adjustment on a graph processor. arXiv. [Preprint.] Mar 30, 2020 [accessed 2024 Aug 27]. Available from: https://arxiv.org/abs/2003.03134.
10. Dally WJ, Keckler SW, Kirk DB. Evolution of the graphics processing unit (GPU). *IEEE Micro* 2021;41:42-51. DOI
11. Gao H, Hu M, Liu Y. Learning driver-irrelevant features for generalizable driver behavior recognition. *IEEE Trans Intell Trans Syst* 2024:1-13. DOI
12. Xue W, Wang H, Roy CJ. CPU-GPU heterogeneous code acceleration of a finite volume Computational Fluid Dynamics solver. *Future Gener Comput Syst* 2024;158:367-77. DOI
13. Wang X, Li Q, Lin Z. On the comparison of mono visual odometry front end in low texture environment. In: 2020 3rd International Conference on Mechatronics, Robotics and Automation (ICMRA); 2020 Oct 16-18; Shanghai, China. IEEE; 2020. pp. 195-200. DOI
14. Ferrer G, Iarosh D, Kornilova A. Eigen-factors an alternating optimization for back-end plane SLAM of 3D point clouds. arXiv. [Preprint.] Sep 4, 2023 [accessed 2024 Aug 27]. Available from: https://arxiv.org/abs/2304.01055.
15. Zheng S, Wang J, Rizos C, Ding W, El-Mowafy A. Simultaneous localization and mapping (SLAM) for autonomous driving: concept and analysis. *Remote Sens* 2023;15:1156. DOI
16. Liu K, Cao M. Dlc-slam: a robust lidar-slam system with learning-based denoising and loop closure. *IEEE/ASME Trans Mechatron* 2023;28:2876-84. DOI
17. Mur-Artal R, Tardós JD. ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras. *IEEE Trans Robot* 2017;33:1255-62. DOI
18. Campos C, Elvira R, Rodríguez JJG, Montiel JMM, Tardós JD. Orb-slam3: an accurate open-source library for visual, visual-inertial, and multimap SLAM. *IEEE Trans Robot* 2021;37:1874-90. DOI
19. Mohammadi MS, Rezaeian M. Towards affordable computing: SiftCU a simple but elegant GPU-based implementation of SIFT. *Int J Comput Appl* 2014;90:30-7. DOI
20. Parker C, Daiter M, Omar K, Levi G, Hassner T. The CUDA LATCH binary descriptor: because sometimes faster means better. arXiv. [Preprint.] Sep 16, 2016 [accessed 2024 Aug 27]. Available from: https://doi.org/10.48550/arXiv.1609.03986.
21. Urban S, Hinz S. Multicol-SLAM-a modular real-time multi-camera SLAM system. arXiv. [Preprint.] Oct 24, 2016 [accessed 2024 Aug 27]. Available from: https://doi.org/10.48550/arXiv.1610.07336.
22. Li J, Deng G, Zhang W, Zhang C, Wang F, Liu Y. Realization of CUDA-based real-time multi-camera visual SLAM in embedded systems. *J Real-Time Image Process* 2020;17:713-27. DOI
23. Ma T, Bai N, Shi W, et al. Research on the application of visual SLAM in embedded GPU. *Wirel Commun Mob Comput* 2021;2021:6691262. DOI
24. Nagy B, Foehn P, Scaramuzza D. Faster than FAST: GPU-accelerated frontend for high-speed VIO. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS); 2020 Oct 24 - 2021 Jan 24; Las Vegas, NV, USA. IEEE; 2020. pp. 4361-8. DOI
25. Zheng M, Zhou S, Xiong X, Zhu J. GPU parallel bundle block adjustment. *Cehui Xuebao/Acta Geod Cartogr Sin* 2017;46:1193-201. (in Chinese) DOI
26. Lu Q, Xu J, Hu L, Shi M. Parallel VINS-Mono algorithm based on GPUs in embedded devices. *Int J Adv Robot Syst* 2022;19:17298814221074534. DOI
27. Cao M, Zheng L, Jia W, Liu X. Fast incremental structure from motion based on parallel bundle adjustment. *J Real-Time Image Process* 2021;18:379-92. DOI
28. Jiang F, Gu J, Zhu S, Li T, Zhong X. Visual odometry based 3D-reconstruction. *J Phys Conf Ser* 2021;1961:012074. DOI
29. Qin T, Li P, Shen S. VINS-Mono: a robust and versatile monocular visual-inertial state estimator. *IEEE Trans Robot* 2018;34:1004-20. DOI
30. Sun H, Zhang Y, Zheng Y, Luo J, Pan Z. G2O-Pose: real-time monocular 3D human pose estimation based on general graph optimization. *Sensors* 2022;22:8335. DOI
31. Sumikura S, Shibuya M, Sakurada K. OpenVSLAM: a versatile visual SLAM framework. In: Proceedings of the 27th ACM International Conference on Multimedia. New York, NY, USA: Association for Computing Machinery; 2019. pp. 2292-5. DOI