Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 6691262, 17 pages
https://doi.org/10.1155/2021/6691262

Research Article

Research on the Application of Visual SLAM in Embedded GPU

Tianji Ma, Nanyang Bai, Wentao Shi, Xi Wu, Lutao Wang, Tao Wu, and Changming Zhao

School of Computer Science and Engineering, Chengdu University of Information Technology, Chengdu Sichuan 610225, China

Correspondence should be addressed to Tianji Ma; 675240786@qq.com

Received 17 December 2020; Revised 10 May 2021; Accepted 26 May 2021; Published 7 June 2021

Academic Editor: Zhili Zhou

In the automatic navigation robot field, robotic autonomous positioning is one of the most difficult challenges. Simultaneous localization and mapping (SLAM) technology can incrementally construct a map of the robot's moving path in an unknown environment while estimating the position of the robot in the map, providing an effective solution for robots to fully navigate autonomously. The camera can obtain corresponding two-dimensional digital images from the real three-dimensional world. These images contain very rich colour, texture information, and highly recognizable features, which provide indispensable information for robots to understand and recognize the environment based on the ability to autonomously explore the unknown environment. Therefore, more and more researchers use cameras to solve SLAM problems, also known as visual SLAM. Visual SLAM needs to process a large number of image data collected by the camera, which has high performance requirements for computing hardware, and thus, its application on embedded mobile platforms is greatly limited. This paper presents a parallelization method based on embedded hardware equipped with embedded GPU. Use CUDA, a parallel computing platform, to accelerate the visual front-end processing of the visual SLAM algorithm. Extensive experiments are done to verify the effectiveness of the method. The results show that the presented method effectively improves the operating efficiency of the visual SLAM algorithm and ensures the original accuracy of the algorithm.

1. Introduction

In order to achieve fully autonomous work in an unknown environment, mobile robots must solve two basic problems of positioning themselves and perception of the environment. Simultaneous localization and mapping (SLAM) was first proposed by Smith and Cheeseman [1] and applied in the field of robotics. It combines the robot's self-positioning and map construction into one. The goal is to make the robot locate itself through the movement of the robot without the prior information of the environment and then establish a real-time map of the environment based on the sensor data; at the same time, the robot's motion trajectory is accurately estimated.

At present, SLAM has relatively mature applications, for example, sweeping robots, drones, Augmented Reality (AR), and Virtual Reality (VR). Autonomous driving and accurate 3D reconstruction are also in rapid development. According to different sensors used, SLAM can be divided into visual SLAM and laser SLAM. Laser SLAM uses LiDAR (Light Laser Detection and Ranging) as a sensor, and the collected data is called point cloud data, which contains accurate angle information and distance information. The distance measurement using LiDAR is more accurate, and the error model is relatively simple. At the same time, LiDAR has the advantages of being insensitive to light. Compared with visual SLAM, laser SLAM's related theoretical research is relatively mature, but the sensors are expensive. Visual SLAM can use a variety of cameras: monocular camera, stereo camera, and depth camera as sensors. These cameras are cheaper than LiDAR and are widely used in various fields of society. At the same time, rich colour information, texture information, and more recognizable image features can be obtained from the images captured by the camera. Therefore, visual SLAM has gradually become the main research direction for solving SLAM problems, but its disadvantage is that real-time processing of a large amount of image data requires high computing resources, which brings a great challenge to real-

time operation on embedded platforms and mobile platforms. Compared with the computing resources of high power consumption PC platforms, embedded platforms and mobile platforms generally have low power consumption, and the computing resources are also greatly restricted. Therefore, it is an important direction of the research to use limited computing resources to efficiently execute algorithms of visual SLAM on embedded platforms.

Thanks to the rapid development of parallel technology, the performance of processors suitable for parallel computing is also rapidly improving, which makes it possible to double the operating efficiency of the algorithm. In recent years, GPU computing performance has achieved rapid growth. Its computing performance, especially parallel computing performance, is far stronger than that of CPU. Researchers have gradually discovered the potential of GPU parallel computing. In order to provide a more friendly interface for researchers and developers to use GPU to solve problems, in 2006, NVIDIA Corporation released CUDA (Compute Unified Device Architecture), a general-purpose parallel computing platform and programming model, as an "engine" to drive GPU to solve complex computing problems, which is more efficient than CPU. After more than ten years of development, CUDA has been widely used in the field of image processing. Huang and Yu [2] used CUDA to accelerate the processing of image segmentation algorithms based on normalization, and Du and Yuan [3] used CUDA to optimize image feature extraction and realized the real-time stitching of panoramic video, which overcomes the shortcomings of high power consumption, nonreal time, and low stability that used to rely on postprocessing.

The current development of visual SLAM has been relatively mature, and there are various types of solutions, including sparse method, semidense method, and dense method, as well as feature point method based on image features and direct method based on image grayscale. The execution efficiency, positioning accuracy, and robustness of these algorithms perform well in specific experimental environments. However, most of these algorithms are performed on desktop-level high-power platforms, and there is very little work to solve visual SLAM problems for embedded platforms. Embedded platforms have many advantages such as low power consumption, miniaturization, low cost, and high reliability, but their performance is far inferior to high-power PC platforms. Due to the fact visual SLAM has high requirements for computing resources and correspondingly high requirements for hardware computing performance, embedded platforms and their performance are easily ignored by visual SLAM researchers.

With the development of embedded hardware, high-performance embedded hardware with integrated GPU has emerged. Since there are a lot of image processing and pose estimation operations in visual SLAM, these operations consume a lot of computing resources. Therefore, GPU parallel computing can be used to accelerate processing. The real-time processing performance of visual SLAM in embedded systems can be effectively improved through the combination of high-performance embedded processing hardware and algorithm optimization, which is beneficial to the mobility, miniaturization, and low energy consumption applications of visual SLAM technology.

In summary, our work mainly studies how to use GPU parallel computing to accelerate processing on embedded hardware to overcome the computational complexity of visual SLAM. Although there have been some works that use GPU to accelerate parallel calculation of certain algorithms in SLAM, Wu et al. [4] and Rodriguez-Losada et al.[5] have implemented beam adjustment and ICP (Iterative Closest Point) algorithms on GPU, but these works are all performed on desktop GPUs.

Figure 1 illustrates a typical visual SLAM system structure diagram, including five parts: visual sensor data, front end (also called visual odometer), back end, mapping, and loop closure detection. The vision sensors input the images, and then, the system performs feature extraction and matching on these input images at the front end, then roughly estimates the position of the feature points and the robot, then transfers the estimated result to the back end, and executes graph optimization to get a more accurate result. In this way, it is possible to locate and then build a map and at the same time transfer the optimized result to the closed-loop inspection to eliminate the accumulated error of the robot moving for a long time and then use the result for tracking. Among these five parts, the front end and the back end are important parts in charge of processing data, and these two parts consume a lot of computing resources.

Our work focuses on the front end, illustrating the main key technologies of feature extraction, feature matching, and the principle of relevant algorithms. We combine the selected scheme and the computing performance of embedded hardware and then select the most appropriate GPU parallel computing method to optimize and improve the visual SLAM processing performance and operating efficiency and ensure good positioning accuracy.

The theoretical basis and related work studied in this paper are as follows.

Firstly, the overall framework of visual SLAM is introduced. The responsibilities and functions of each module in the framework are described.

Then, the main method of the visual SLAM front end, the feature point method, is introduced. In this paper, we use ORB (Oriented FAST and Rotated BRIEF) features as the front-end implementation method which has the fastest calculation speed on the basis of meeting the accuracy of feature detection, to ensure the fast processing of the embedded platform. As for the relevant visual SLAM algorithm, we choose ORB-SLAM2 [6], a visual SLAM algorithm that uses ORB features for detection and matching in the visual front end.

Finally, the parallel mechanism is analysed on the CUDA-based GPU hardware architecture and programming model. On this basis, the detailed parallel analysis of the key technologies of the selected scheme is carried out. A reasonable parallelization scheme was designed, and the GPU parallelized visual SLAM system was built on embedded development board NVIDIA Jetson TX2.

In order to evaluate the performance of the system, relevant experiments were carried out on the datasets. The results show that the whole system is in good working

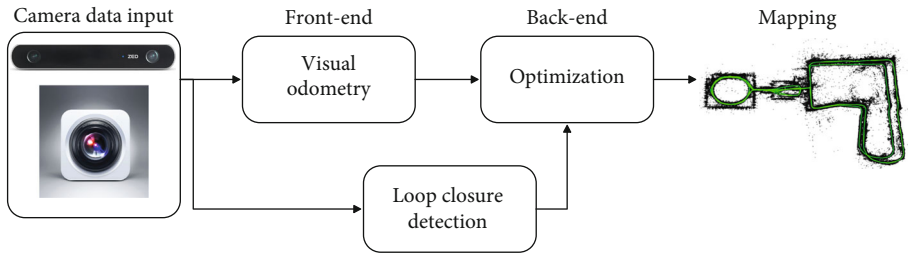


Figure 1: A block diagram of a typical visual SLAM system.

condition. In addition, by counting the time overhead of executing datasets, it is intuitively shown that the use of GPU parallelization effectively improves the operating speed of the visual SLAM system on the embedded platform.

2. Front-End Visual Odometry

The front end is at the lower level in the visual SLAM system, also known as visual odometry (VO) [7]. For visual odometry, its focus is on the frame-to-frame motion between adjacent images. When the sensor data module transmits the image frame sequence (i.e., video stream) to the visual odometry, its function is to extract the key information of adjacent image frames to roughly estimate the camera movement in advance to provide better results for the back end. At present, there are two main methods of visual odometry: feature point method and direct method. In this paper, we use the feature point method.

2.1. Feature Point Method. The front end based on the feature point method is a classic method of visual odometry. It uses the redundancy of the image to detect and extract feature points from the preprocessed input image and then performs feature matching to estimate the camera motion trajectory. Therefore, it avoids processing the complete image containing a large amount of redundant information and greatly reduces the amount of calculation while preserving the important information of the image. And it runs stably and is not sensitive to lighting and dynamic objects, so it is widely used in visual SLAM. For the visual odometry of the feature point method, one of the keys is to use feature detection algorithms to extract the best features from a frame of images. At present, the development of image feature detection algorithms is relatively mature. Commonly used feature detection algorithms include SIFT [8], SURF [9], ORB, and AKAZE [10]. For details, please check the relevant literature.

2.2. ORB Feature Detection Algorithm. The ORB algorithm was proposed by Rublee et al. [11] in 2011. It combines an improved FAST (Features from Accelerated Segment Test) corner detection algorithm and a direction normalized BRIEF (Binary Robust Independent Elementary Features) feature descriptor algorithm. The ORB feature detector will detect FAST corner points in each layer of the image Gaussian pyramid and use Harris corner scores to evaluate the detection points to select the highest-quality feature points. Since the original BRIEF feature descriptor is very sensitive to rotation, the ORB algorithm is improved. ORB features have scale invariance, rotation invariance, and certain affine invariance.

3. GPU Parallel Accelerated Visual SLAM

3.1. GPU Hardware Features. In recent years, with the rapid development of science and technology, the problems faced by many research fields have become larger and the corresponding requirements for computing performance have become higher and higher. Even if CPU manufacturers represented by Intel and AMD have introduced multicore architecture CPU to make up for the limit of single-core performance improvement, their performance still cannot meet the needs of the market. For the GPU, driven by the market urgent need for real-time and high-definition 3D image rendering, GPU has gradually developed into a highly parallel, multithreaded, multicore processor architecture like today, with huge computing power and extremely high memory bandwidth.

The reason behind this huge computing performance gap is the difference in hardware structure between GPU and CPU. First of all, the CPU is composed of several cores optimized for sequential serial processing. While the GPU is composed of thousands of smaller, more efficient cores, which are specifically designed to handle multiple tasks at the same time and can efficiently handle parallel tasks.

It can be seen from the structure (see Figure 2) that a large part of the CPU is used for caching and control, and there are relatively few arithmetic logic units, while the GPU is the exact opposite, and the computing units occupy the vast majority.

In the early days, it was very inconvenient for researchers to use GPU to perform calculations in the field of non-graphics rendering, because GPU is dedicated to graphics rendering and has streamlined rendering pipelines. Therefore, general-purpose computing programs can only be encapsulated into rendering programs and embedded in these pipelines before they can be executed by the GPU. With the increasing demand for general-purpose computing, in order to provide researchers and developers with a more friendly interface to use GPU to solve problems, NVIDIA Corporation released CUDA in 2006, a general-purpose parallel computing platform and programming model, as an "engine" to drive the GPU to efficiently solve complex computing problems. Now, this kind of general-purpose parallel computing is widely used in various industries and fields, including deep learning that has been developed rapidly in recent years.

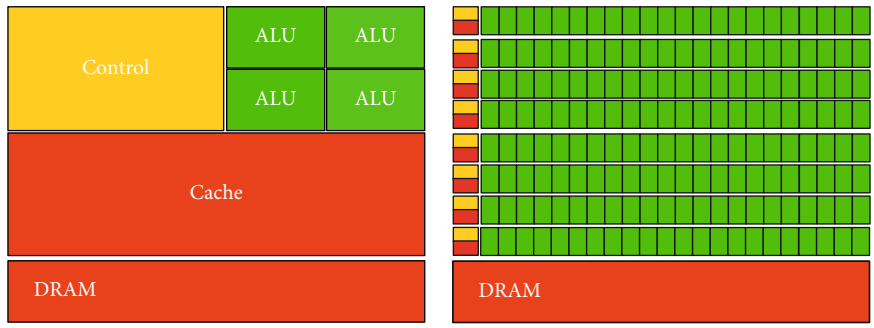


Figure 2: Diagram of CPU and GPU structure: (a) CPU structure; (b) GPU structure.

3.2. CUDA Hardware Model. A simplified diagram of the GPU hardware architecture that supports CUDA is shown in Figure 3. The most basic processing unit is the Streaming Processor (SP), also known as CUDA-CORE, which is responsible for the execution of each specific instruction. The GPU parallel computing is essentially a large number of SP simultaneous processing tasks. The core unit is the streaming multiprocessor (SM), also known as the GPU core, which consists of multiple SPs, thread schedulers, memories, and other units. The number of SMs owned by different models of GPU and the number of SPs contained in each SM are different. Therefore, a GPU may have thousands of SPs. In theory, these SPs can execute instructions at the same time, so the calculation speed is very fast.

3.3. CUDA Programming Model. For the typical CUDA program execution procedure, see Figure 4. The CUDA parallel program executed on the GPU is also called a kernel function, which is specifically used to complete GPU parallel computing tasks.

The program is first executed from the host side, the serial program performs initialization work, and the kernel function is started after the data and storage space required for the execution of the kernel function are allocated. And then the device side will generate a large number of threads based on various variable parameters set in the kernel function, and these threads will be organized into thread blocks. Subsequently, the thread blocks will be allocated to the SM for parallel execution. Each thread block is divided into several groups of threads when executed on the SM, and each group of threads will eventually be mapped to a group of SPs in the SM for parallel calculation. After the kernel function is executed, the serial program will copy the calculation result from the device to the host. Then, prepare for the next execution of the kernel function, or complete and end the execution of this CUDA program.

3.4. Experimental Hardware. This time, the high-performance embedded platform we used is Jetson TX2. It is a powerful multicore mobile SOC released by NVIDIA in 2017, mainly for smart terminal devices such as smart robots, drones, unmanned driving, smart cameras, and portable medical equipment. Its CPU has a total of 6 cores, including 4 Cortex-A57 and 2 customized Denver cores. In addition, TX2 is a heterogeneous system, which integrates a Pascal architecture GPU with 256 CUDA cores. For details of technical specifications, please refer to the official website [12]. Its main technical specifications are shown in Table 1. The computing performance and related parameters of its GPU are shown in Table 2. Our experimental software environment includes Ubuntu 18.04, ROS (Robot Operating System) Melodic, CUDA 10.0, OpenCV 3.4.1, Eigen 3.3.3, Ceres-solver 1.13.0, and Python 2.7.17.

In comparison to other boards with a similar form size, Figure 5 [13] shows the total runtime of the C-Ray benchmark program with the Jetson TX2.

3.5. Front-End Parallelization. For the front end based on the feature point method, the main processing procedures are image feature extraction and matching. They consume more than half of the computing resources and are calculated for images, so this part is particularly suitable for parallelization. This paper will focus on the following areas of research: feature extraction parallelization and feature matching parallelization. We will analyse how the procedures of image feature extraction and matching can be parallelized. Then, we parallelize the relevant parts by CUDA and finally test the execution efficiency of GPU parallelization through experiments.

3.5.1. Parallelization of Feature Extraction. After the front end obtains a frame of images transmitted from the visual sensor, it constructs an image Gaussian pyramid based on the original image first. Afterwards, the feature points and feature vectors are extracted from each image layer of the image pyramid to ensure that the ORB features are scale-invariant. Finally, all feature points and feature vectors extracted from each image layer will be mapped to the original image, but this makes the image features of each original image too dense and repetitive. Therefore, it is necessary to delete the repeated feature points and perform nonmaximum suppression on the rest of feature points to ensure that the distribution of the feature points is relatively uniform and to improve the effect of image matching. The main calculation procedure of the ORB feature extraction (see Figure 6) is as follows:

(1) Construct an image Gaussian pyramid

(2) Perform FAST feature point detection in each image layer of the pyramid

(3) Perform coordinate normalization in image layers of different sizes

(4) Delete duplicate FAST feature points. Compare each feature point with the corresponding feature point in the upper and lower adjacent image layers at the same scale, and keep the feature point with the largest response value

(5) Nonmaximum suppression: each feature point is compared, respectively, with 26 adjacent points in the image layer where it is located and in the upper and lower adjacent image layers at the same scale. Only when the response value of this feature point is greater than the other feature points will it be kept; otherwise, it will be deleted

(6) Sort feature points according to FAST and Harris response values [14]. Select the top N best feature points, and the N value is preset according to requirements

(7) Assign the direction to each feature point, and calculate the BRIEF descriptor to complete the extraction of ORB features

Using parallelism analysis of the ORB feature extraction process, we can get the conclusion:

(1) In the FAST feature point detection procedure, there is no data communication between the image layers of the Gaussian pyramid, so FAST feature point detection can be performed in parallel in each image layer

(2) FAST feature point detection only has data association with each pixel and its neighbouring pixels in the image, and the detection procedure is exactly the same, so it can be executed in parallel on a large scale

(3) For each feature point, the calculation of feature orientation and feature description is based on the fact that the feature point has a data association with separate local image information in the image in which it is located and can therefore be calculated in parallel

Based on the above analysis, we can roughly describe the feature extraction parallelization steps. First, the Gaussian pyramid of the image is constructed on the GPU. Then, based on the Gaussian pyramid of the image, the feature detection, feature orientation calculation, and feature description calculation are performed on the GPU. Finally, the feature point information and feature descriptions of the image are used as the output of the ORB feature extraction algorithm. The framework of the feature extraction parallelization algorithm is shown in Figure 7.

As the program on the GPU and CPU is operated asynchronously, in order to parallelize the feature extraction algorithm, the algorithm is optimized in two ways: by adjusting the task allocation to decrease the idle time of the GPU and by adjusting the thread allocation to increase the usage of the streaming multiprocessor (SM), thus solving the problem of imbalanced load on the GPU computing resources.

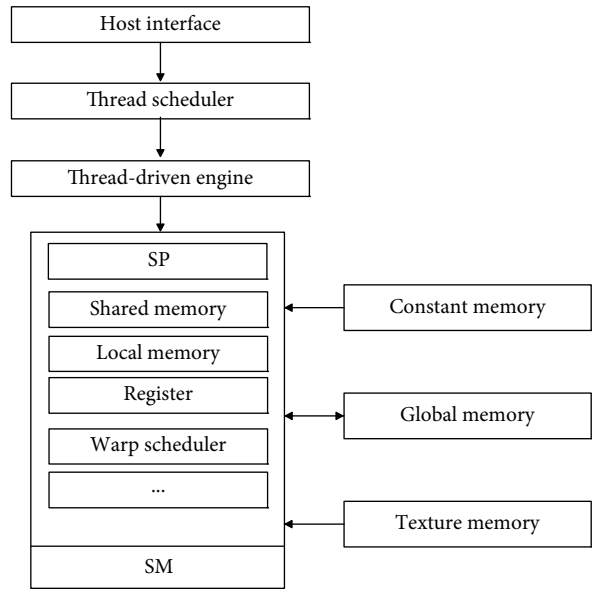


Figure 3: GPU hardware architecture diagram.

Adjustment of Task Allocation. In the feature extraction algorithm, the image Gaussian pyramid is constructed and then feature detection is performed based on the constructed Gaussian pyramid. When operating feature detection, the GPU synchronization instruction needs to be called first to ensure that the GPU has completed the construction of the Gaussian pyramid, and if the GPU has not completed the computation, the CPU will wait until the GPU has finished processing. This paper solves the problem of idle CPU waiting by adjusting the call of the synchronization instruction to improve the computational efficiency of the feature extraction.

When performing feature detection, each layer of the Gaussian pyramid is usually processed separately, where the original image is the first layer, followed by the scaled-down scale image. However, when performing feature detection on the first layer of the Gaussian pyramid, i.e., the original image, the results of the Gaussian pyramid construction calculation are not required. Therefore, in this paper, after operating the built Gaussian pyramid instruction, the feature detection is performed on the original image first, and then, the GPU synchronization instruction is called after the feature detection of the original image is completed; then, the completed Gaussian pyramid is used to continue the feature detection task, thus avoiding the time spent waiting for the GPU to complete the computation task. By adjusting the GPU synchronization instructions, the validity of the data is ensured and the waiting time of the CPU is decreased, thus improving the computational efficiency of the feature extraction algorithm. The algorithm is shown in Algorithm 1.

After feature detection is complete, nonmaximum suppression is performed to remove some of the feature points

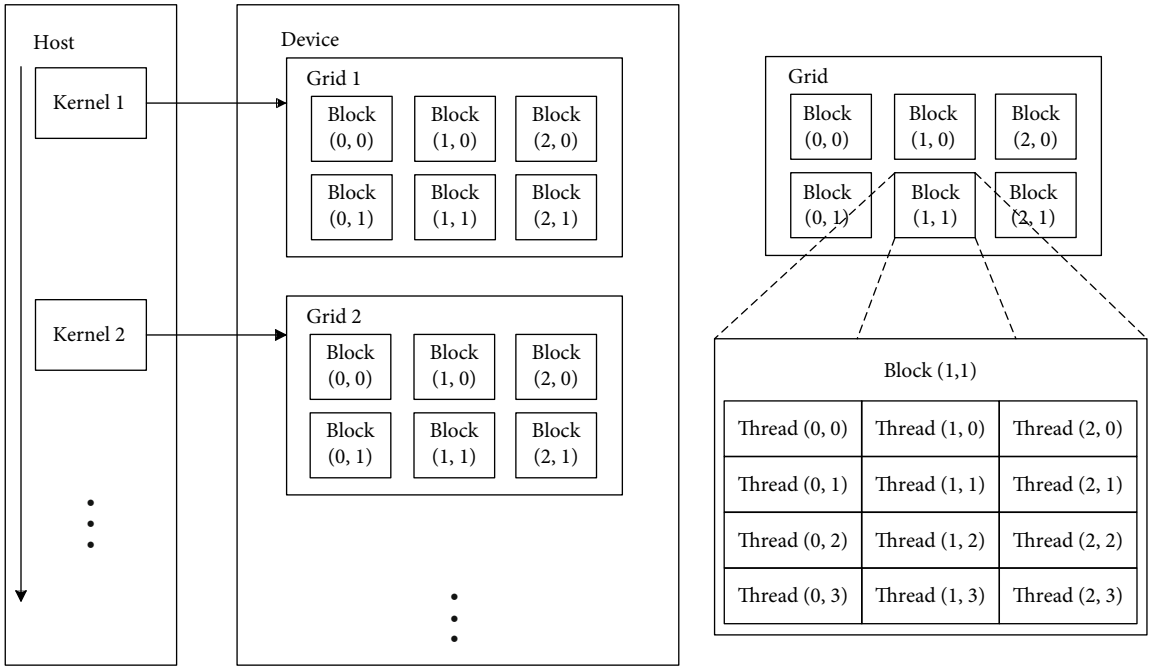


Figure 4: Host refers to the host side, representing the CPU, and device refers to the device side, representing the GPU. Grid is the outermost thread organization structure in CUDA.

Table 1: TX2 main performance indicators.

Parameter	Technical specifications
CPU	ARM Cortex-A57 (quad-core) 2 GHz+NVIDIA Denver2 (dual-core) 2 GHz
GPU	256 CUDA-CORES Pascal 1.30 GHz
Memory	8 GB 128-bit LPDDR4 1866 MHz
Storage	32 GB eMMC 5.1
TDP	7.5 W
Size	87 mm × 50 mm

Table 2: GPU-related parameters of TX2.

Parameter	Technical specifications
Computing performance version	6.2
Maximum number of threads in the thread block	1024
Maximum dimension of the thread block	3
Maximum dimension of the thread block in the x and y direction	1024
Maximum dimension of the thread block in the z direction	64

to ensure that they are evenly distributed across the image. Since nonmaximum suppression has a lot of branches and loops, which are not suitable for operation on the GPU, the CPU is still used for nonmaximum suppression. However, this causes the GPU to be idle all the time while nonmaximum suppression is performed.

To improve the resource usage of the GPU, we adjusted the task allocation for feature detection and nonmaximum suppression. We note that although nonmaximum suppression uses the results of feature detection, there is no correlation between the data at different layers in the Gaussian pyramid. Therefore, we adjusted the execution order of feature detection and nonmaximum suppression. We use the GPU to perform feature detection on the current layer image while using the CPU to perform nonmaximum suppression of feature points on the previous layer image. This approach did not lead to data conflicts, while improving the resource usage of the GPU. The algorithm is shown in Algorithm 2.

Increase the usage of the streaming multiprocessor. According to the above CUDA implementation principle, when executing CUDA programs in the GPU, computational resources need to be allocated by setting the size of thread

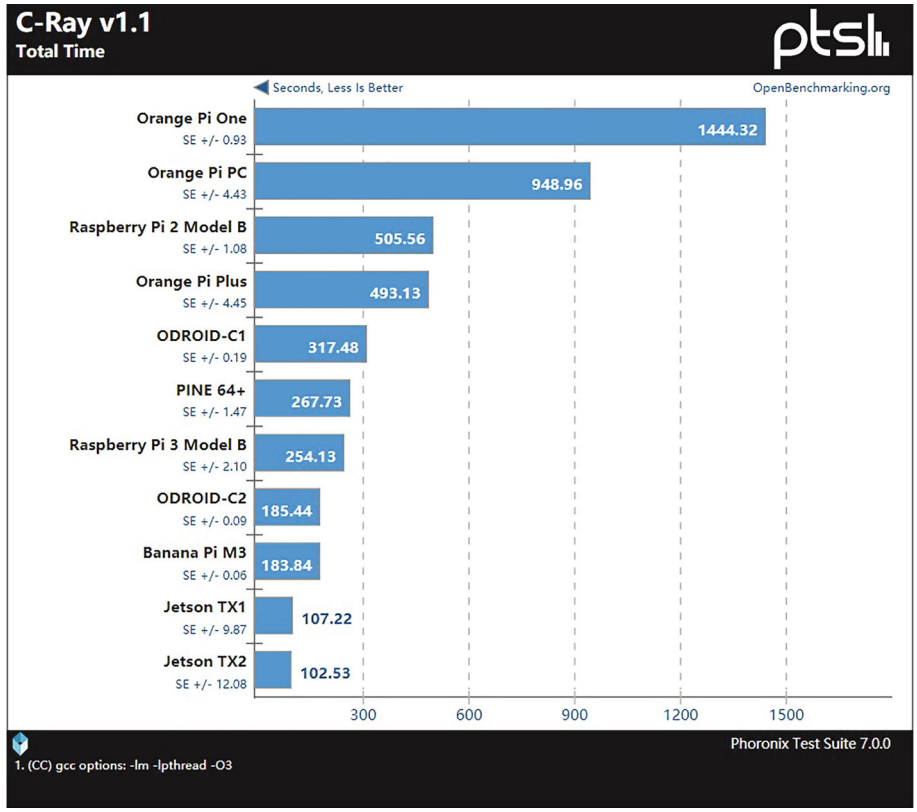


Figure 5: Comparison of Raspberry Pi 3 with other ARM Linux boards with C-Ray multithreaded ray tracer.

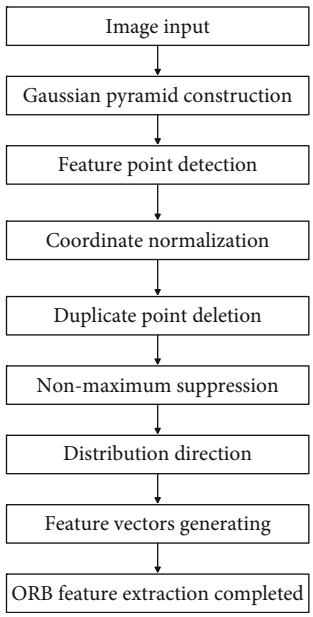


Figure 6: ORB feature extraction flowchart.

blocks and thread grids, and after that, the threads are loaded by warp into the computational core SM of the GPU for processing. Depending on the size of the image data being processed, the computational resources are reallocated to increase the usage of the SM.

Performing Feature Detection. In order to allocate as many threads as possible when processing images at different scales, 1×4 pixels are allocated to each thread for computation in feature detection. Also, the thread block is set to two dimensions, with a size of 32×8, and the thread grid is set to two dimensions.

$$N_{\text{grid\_1}} = \left\lceil \frac{\text{imgcol}}{\text{block}.x} \right\rceil \times \left\lceil \frac{\text{imgrow}}{\text{block}.y} \right\rceil, \tag{1}$$

where $N_{\text{grid\_1}}$ is the size of the thread grid, imgrow is the height of the image, imgcol is the width of the image, and block.x and block.y are the size of the thread block. Allocating more threads in the thread block can reduce the idle wait time of the SM when accessing global memory. However, allocating too many threads will result in the number of registers being so difficult to meet the computational demands that the SM will transfer some of the data stored in the registers to global memory with higher transfer latency. In addition, allocating too many threads will also cause calls to GPU core functions to fail. After testing, this paper allocates as many threads as possible in the thread block, i.e., 32×8, while ensuring stable system operation. On this basis, the size of the thread grid is set according to the size of the image and the thread block.

Calculation of Feature Orientation. Since the calculation of feature orientation requires the points in the area around the feature point to be involved in the calculation, 32 threads are allocated to calculate the orientation information for each

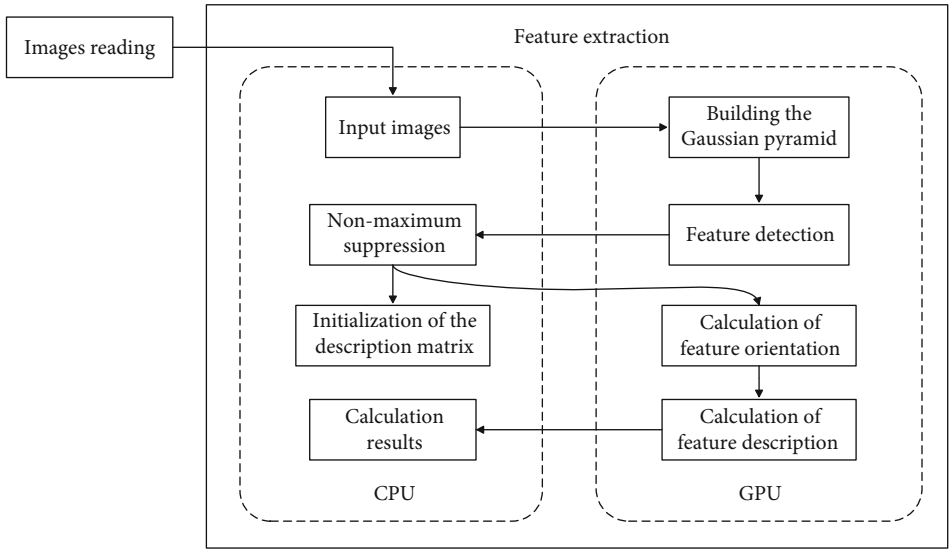


Figure 7: The framework of the feature extraction parallelization algorithm.

Input: An image I_m, Gaussian pyramid layers levelNum
Output: A set of feature points of the first layer image
1 Image is transferred to the GPU memory
2 cudaMemCopyHostToDevice(I_m);
3 Construct Gaussian pyramid of the image on GPU
4 for i ⟵ 0 to levelNum
5 d_ I_p[i] ⟵ gpuBuildScalePyramid(I_m);
6 end
7 Calculate the feature points of the first layer image
8 gpuFeatureDetect(d_I_p[0]);
9 P_t[0][0…N_{t1}] ⟵ gpuGetFeaturePoint();
10 Synchronize the instructions of constructing Gaussian pyramid
11 synchronization();
Note: The beginning of d refers to the data allocated in the video memory, N_{t1} is the temporary number of feature points

Algorithm 1: Optimization of Gaussian pyramid construction and feature detection algorithm processes.

Input: Image Gaussian pyramid Collection d_I_p[0…levelNum], Gaussian pyramid layers levelNum
Output: A set of feature points
1 Perform feature detection and nonmaximum suppression alternately
2 for i ⟵ 1 to levelNum
3 gpuFeatureDetect(d_I_p[i]);
4 P_t[i-1][0…N_{t2}] ⟵ nonmaximumSuppression(P_t[i-1][0…N_{t1}]);
5 P_t[i][0…N_{t1}] ⟵ gpuGetFeaturePoint();
6 end
7 P_t[levelNum][0…N_{t2}] ⟵ nonmaximumSuppression(P_t[levelNum][0…N_{t1}]);
Note: The beginning of d refers to the data allocated in the video memory, N_{t1} and N_{t2} are the temporary number of feature points

Algorithm 2: Optimization of feature detection algorithms and nonmaximum suppression algorithm processes.

feature point. Also, the thread block is set to two dimensions, its size is 32×8, and the thread grid is set to one dimension.

$$N_{\text{grid\_2}} = \left\lceil \frac{\max\{n\text{point}_i\}}{\text{block}.y} \right\rceil, \quad (2)$$

where $N_{\text{grid\_2}}$ is the size of the thread grid and $\max\{n\text{point}_i\}$ is the maximum number of feature points in the different images. As the number of feature points is not exactly the same across images, to facilitate the allocation of computational resources, we allocate directly based on the maximum number of feature points in each image. For the part with

```
Input: Feature point information in point cloud maps inInf[0…pointNum]
Output: Selected feature point information outInf[0…pointNum], Number of feature points selected nSelection
1   cudaMemCopyHostToDevice(d_inInf[0…pointNum], inInf[0…pointNum]);
2   for i ⟵ 0 to pointNum
3      if(gpuIsSelection(d_inInf[i]))
4         d_outInf[i].flag=0;
5         d_outInf[i] ⟵ gpuSaveSelectedPointsInformation();
6      end
7   end
8   cudaMemCopyDeviceToHost(outInf[0…pointNum], d_outInf[0…pointNum]);
9   nSelection=0;
10   for i ⟵ 0 to pointNum
11      if(outInf[i].flag==0)
12         nSelection++;
13      end
14   end
Note: The beginning of d refers to the data allocated in the video memory, pointNum is the number of feature points in the point cloud map
```

Algorithm 3: Local point selection algorithm.

Table 3: The average time overhead of the execution dataset.

Dataset	The average time overhead (seconds)		
	Intel Core i5-8300H virtual machine on a laptop	TX2 without GPU acceleration	TX2 with GPU acceleration
MH_01_easy	314.2	469.4	302.2
MH_02_easy	271.4	388.2	243.6
MH_03_medium	242.0	343.2	217.4
MH_04_difficult	172.8	251.6	152.6
MH_05_difficult	191.8	278.8	165.8

Table 4: The number of image frames contained in each dataset.

Dataset	The number of image frames (frames)
MH_01_easy	3640
MH_02_easy	3000
MH_03_medium	2640
MH_04_difficult	1980
MH_05_difficult	2220

Table 5: The average execution efficiency of the execution dataset.

Dataset	The average execution efficiency (frames per second)		
	Intel Core i5-8300H virtual machine on a laptop	TX2 without GPU acceleration	TX2 with GPU acceleration
MH_01_easy	11.59	7.76	12.05
MH_02_easy	11.05	7.73	12.32
MH_03_medium	10.91	7.69	12.14
MH_04_difficult	11.46	7.87	12.98
MH_05_difficult	11.58	7.96	13.39

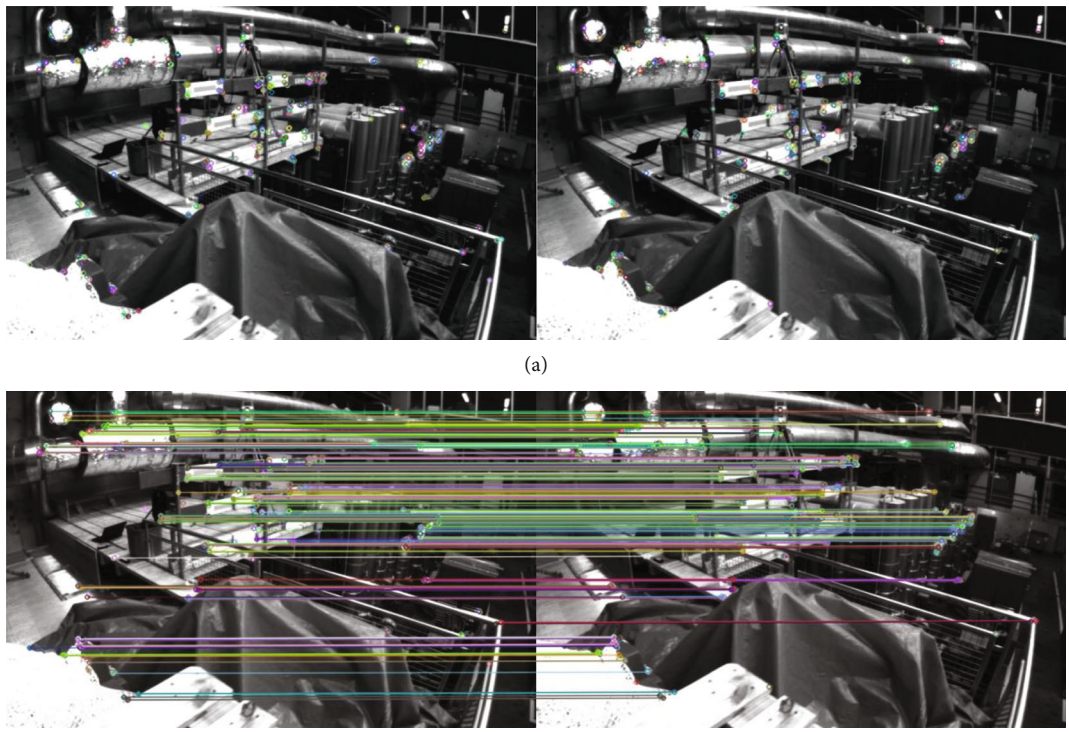
fewer feature points, the computation is kept consistent by directly allocating empty threads. Also, to differentiate the computational results, the feature orientation of different images is computed in different dimensions of the thread grid.

Calculating Feature Descriptor. The feature descriptor for each feature point consists of 32 bytes, which can be calculated separately using 32 threads. Also, the thread block is set to two dimensions, its size is 32×8, and the thread grid is set to one dimension.

$$N_{\text{grid\_3}} = \left\lceil \frac{\max\{n\text{point}_i\}}{\text{block}.y} \right\rceil, \tag{3}$$

where $N_{\text{grid\_3}}$ is the size of the thread grid. The size of the feature descriptor is identical to the number of feature points, so the same computational resource allocation scheme is used as for computing the feature direction: 32 threads are used to compute the feature descriptor for a feature point and 8 feature points are allocated to a thread block; then, the size of the thread grid is set according to the size of the feature descriptor and the thread block.

3.5.2. Parallelization of Feature Matching. In addition to feature extraction, feature matching is another relatively time-consuming part of the visual SLAM algorithm based on the



(a)

(b)

Figure 8: Feature extraction and matching of two adjacent frames. (a) Feature extraction. The coloured circles in the two images indicate ORB feature points. (b) Feature matching.

Table 6: Correlation data for feature extraction and matching in two images.

Test items	CPU-only	GPU-accelerated
Number of features extracted 1	976.4	973.2
Number of features extracted 2	962.3	955.6
Number of successful matches	120.7	118.7
Number of false matches	20.3	18.6

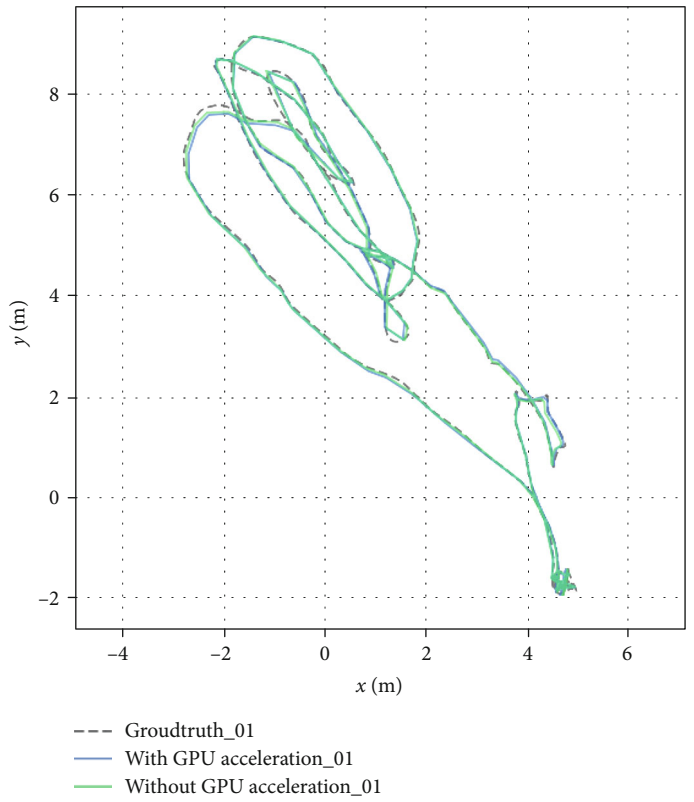
Table 7: Comparison of average execution times for parallelization.

Test items	CPU-only (s)	GPU-accelerated (s)
Feature extraction 1	0.0283	0.0126
Feature extraction 2	0.0275	0.0122
Feature matching	0.0153	0.0072
Total time consumed	0.0711	0.0320

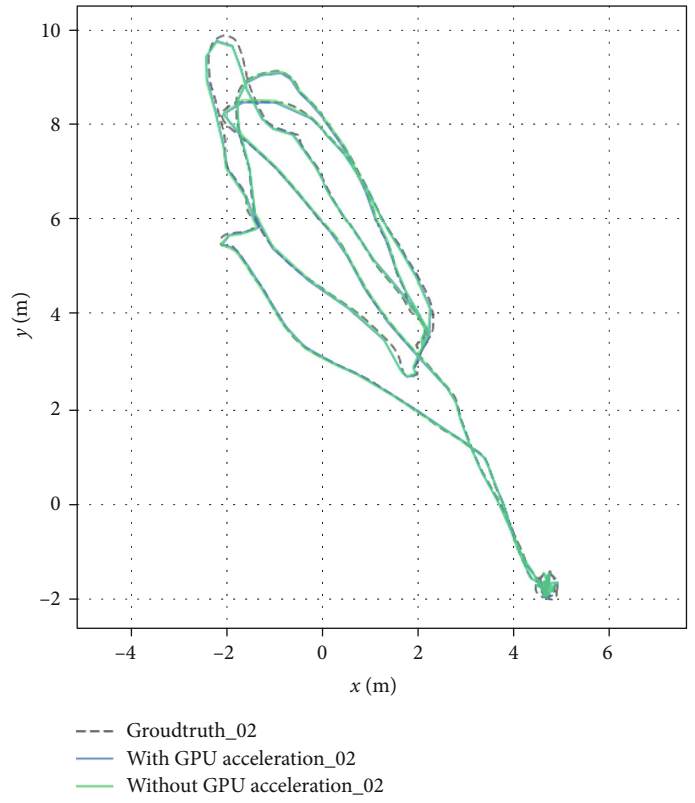
feature point method. Feature matching means matching feature points in the current frame with feature points in the point cloud map and calculating the current pose based on the matching result. As the SLAM algorithm operates, the number of feature points in the point cloud map will gradually increase, which will cause feature matching to take longer and longer. In order to reduce the computation time for feature matching, local point selection is performed before feature matching to reduce the number of feature points in the point cloud map during feature matching. However, as the number of feature points in the point cloud map increases, the computation time for local point selection will also increase. Therefore, this section improves the computational efficiency of the algorithm by accelerating the local point selection algorithm in feature matching in parallel.

To improve the computational efficiency of local point selection, a parallel algorithm for local point selection is implemented on the GPU. Due to the simplicity of the algorithm process for local point selection, it can simply be rewritten in CUDA. Also, both the thread block and thread grid are set to one dimension, with a thread block size of 256 and a thread grid size of pointNum/256, where pointNum is the number of feature points in the point cloud map. In addition, we save and pass back to memory the number of selected feature points and the selected feature points. The algorithm is shown in Algorithm 3.

3.6. Front-End Parallelization Efficiency Test. In order to verify that the GPU parallelization method used in this paper can effectively accelerate the processing speed of the front end, the parallelization method is tested on the EuRoC MAV Datasets [15]. For details of these datasets, please refer to the document [16]. In the datasets, those titles started with MH are the videos recorded indoors. The environment was complicated, which makes the dataset challenging to process. The word, such as easy, medium, and difficult, in those titles, respectively, represents the complexity of the scene in the video. As the complexity increases, fast motion scenes, scenes with dramatic lighting changes, and scenes with the camera

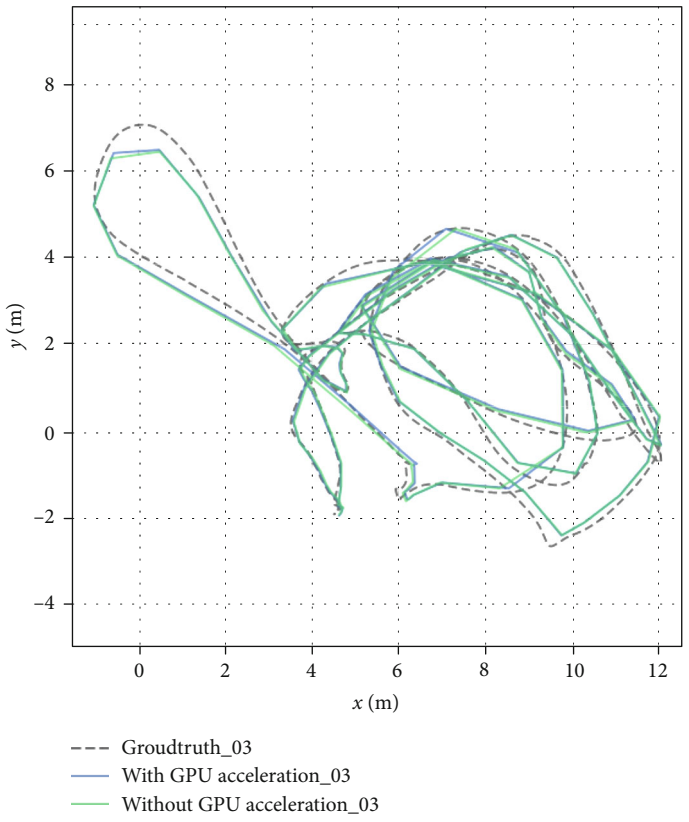


(a)

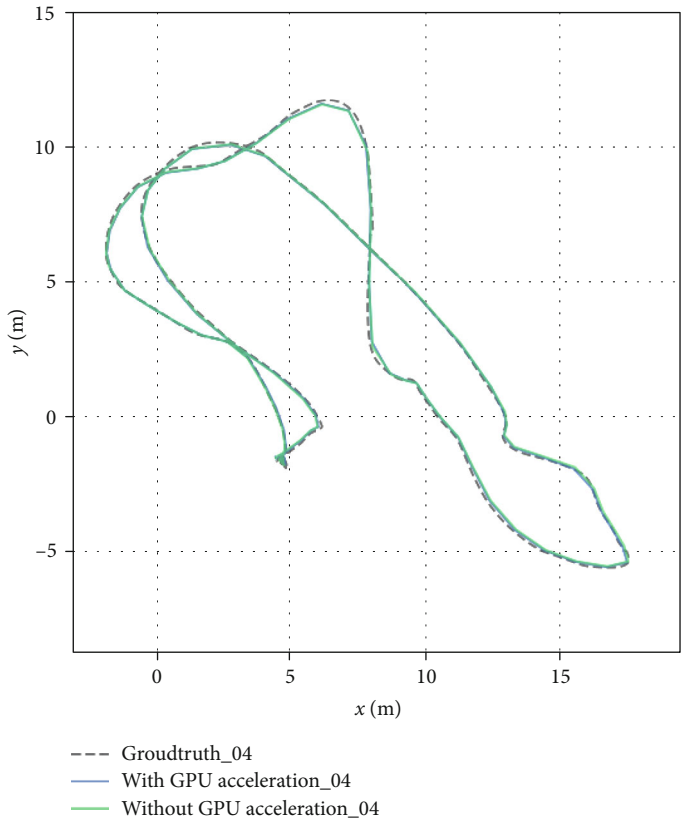


(b)

Figure 9: Continued.

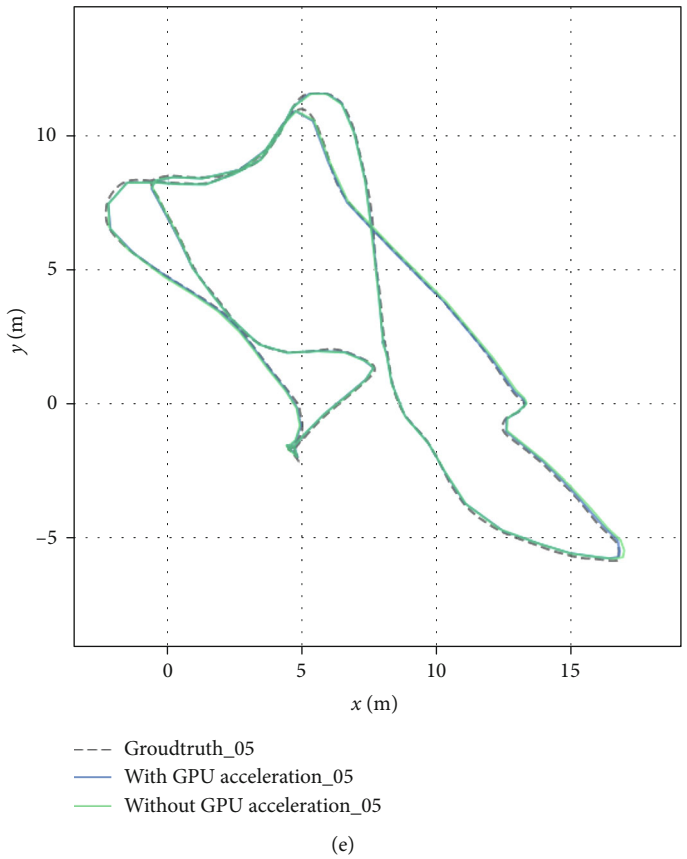


(c)



(d)

Figure 9: Continued.



(e)

Figure 9: The drafting results of the camera motion trajectory in each dataset: (a) MH_01_easy; (b) MH_02_easy; (c) MH_03_medium; (d) MH_04_difficult; (e) MH_05_difficult.

Table 8: The absolute error between the trajectory estimated by the visual SLAM algorithm without GPU acceleration and the actual value.

Dataset	Without GPU acceleration			
	Mean (m)	Median (m)	RMSE (m)	STD (m)
MH_01_easy	0.228	0.225	0.256	0.116
MH_02_easy	0.280	0.272	0.308	0.129
MH_03_ medium	0.423	0.397	0.492	0.250
MH_04_difficult	0.424	0.417	0.489	0.243
MH_05_difficult	0.409	0.387	0.477	0.246

Table 9: The absolute error between the trajectory estimated by the visual SLAM algorithm with GPU acceleration and the actual value.

Dataset	With GPU acceleration			
	Mean (m)	Median (m)	RMSE (m)	STD (m)
MH_01_easy	0.249	0.245	0.284	0.136
MH_02_easy	0.284	0.278	0.313	0.132
MH_03_ medium	0.431	0.401	0.499	0.252
MH_04_difficult	0.448	0.439	0.513	0.250
MH_05_difficult	0.421	0.396	0.487	0.245

fast turning will appear in the video. These scenes will affect the accuracy of the estimated camera movement trajectory. During the experiment, the average time cost of executing each set of datasets was, respectively, recorded under the conditions of using Intel Core i5-8300H virtual machine on a laptop computer, using Jetson TX2 without GPU acceleration and using Jetson TX2 with GPU acceleration. Among them, the virtual machine has an 8-core processor, its rated operating frequency is 2.30 GHz, the memory size is 8 GB, and there is no virtual graphics card. The average time is obtained by averaging 5 times of testing. The recorded results are shown in Table 3.

From the results, the average time cost of running the dataset after using GPU parallelization on TX2 is greatly reduced. The time cost was roughly reduced by a third. Even its average time overhead is less than the virtual machine used in the test. Therefore, the use of GPU parallelization can significantly improve the processing performance of the embedded platform on the visual SLAM front end.

According to the information provided by the official website [15] of the datasets, the official staff used stereo cameras when recording the dataset, and the frequency of capturing images was 20 frames per second. According to Table 2 in the document [16], we can know the duration of each dataset. Therefore, we can draw Table 4.

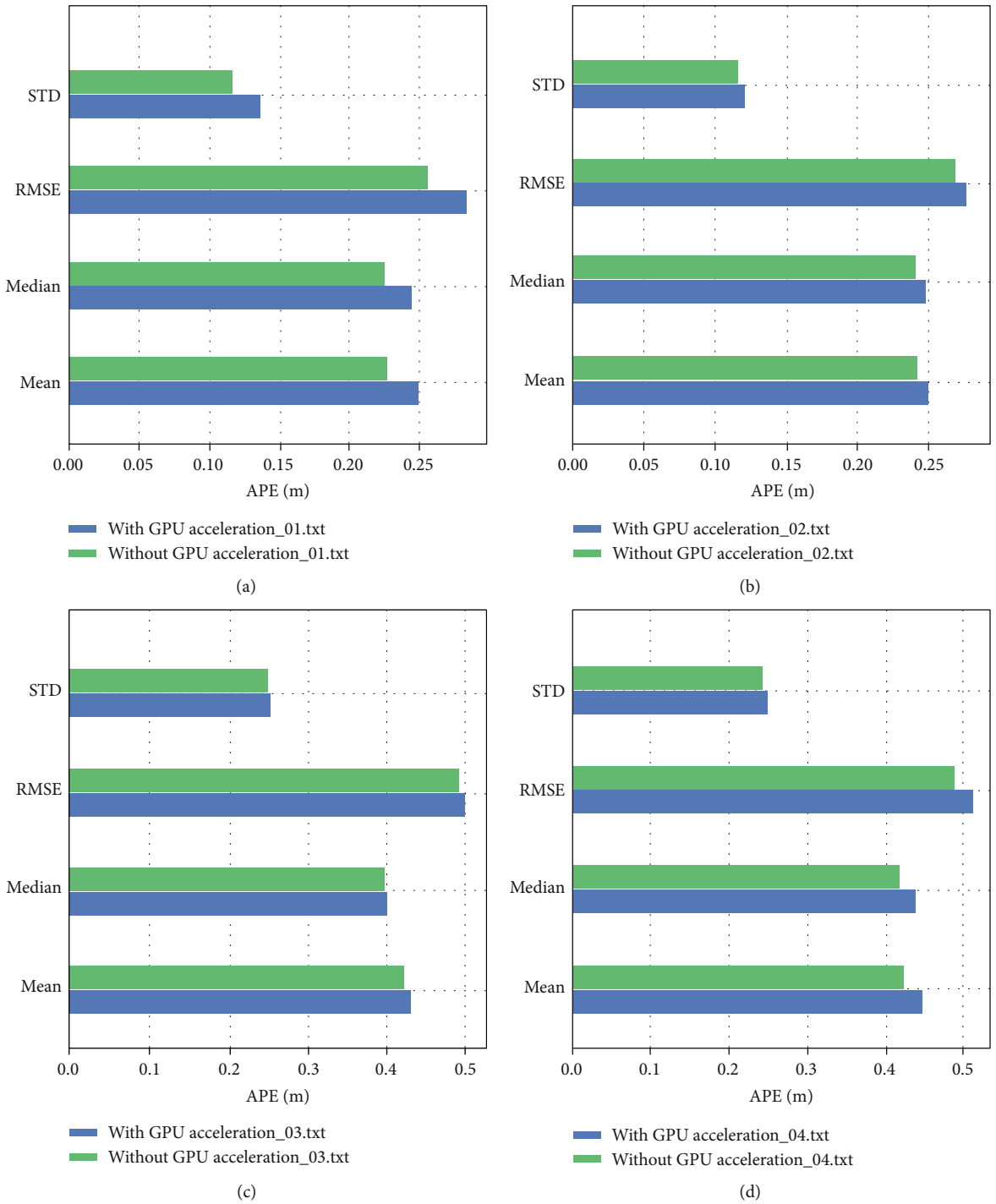
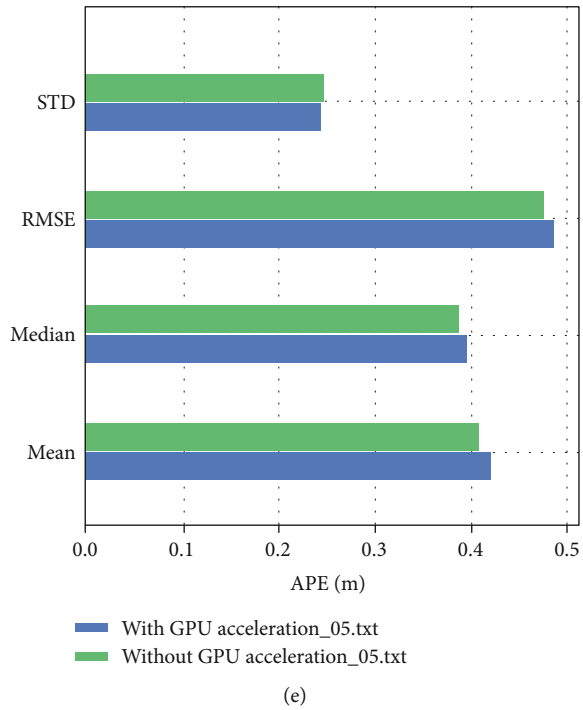


Figure 10: Continued.



(e)

Figure 10: The absolute error between the two trajectories estimated and the actual value: (a) MH_01_easy; (b) MH_02_easy; (c) MH_03_medium; (d) MH_04_difficult; (e) MH_05_difficult.

According to Tables 3 and 4, we can know how many image frames can be processed per second. The results are shown in Table 5.

From Table 5, we can conclude that compared with the other two cases, after using GPU parallelization, the execution efficiency of the visual SLAM front-end algorithm on TX2 is the highest.

In addition, for efficiency tests before and after front-end parallelization, two adjacent frames from the dataset were intercepted for further experiments (see Figure 8), comparing the average number of feature extractions and successful matches and the average execution time of these two processes, respectively.

For the experiments, the data related to feature detection and matching of the two images were counted under both CPU-only and GPU-accelerated condition, including the number of features extracted per image, the number of features successfully matched between the two images, and the number of false matches (see Table 6). The final results of the experiment were obtained by averaging the results obtained from five replications.

Accordingly, the time overheads for feature detection and matching of the two images, including the time spent on feature extraction per image, the time spent on image feature matching, and the total time consumed, are recorded for both CPU-only and GPU-accelerated conditions, respectively (see Table 7) (unit in seconds).

From the experimental results, it can be seen that whether GPU parallelization is used or not does not have a direct impact on the number of image features extracted, the number of successful matches per group of images, or the number of false matches but has a direct and obvious impact on its execution time. From the results, the time overhead of the feature extraction and matching process is greatly reduced after using GPU parallelization on TX2, reducing the time by about half, so using GPU parallelization significantly improves the processing performance of the visual SLAM front end.

3.7. Movement Trajectory Estimation Test. Under the conditions of using Jetson TX2 without GPU acceleration and using Jetson TX2 with GPU acceleration, respectively, the camera motion trajectory was estimated through the visual SLAM algorithm. We separately estimated the camera motion trajectory in these five datasets. After that, we used evo [17], a Python package for the evaluation of odometry and SLAM contributed by Grupp et al., to draft, evaluate, and compare the trajectory output of odometry and SLAM algorithms. The drafting results of the camera motion trajectory in datasets are shown in Figure 9. In the five figures, the dotted line represents the actual value of the camera's motion trajectory, the blue line represents the estimated value of the trajectory when GPU acceleration is enabled, and the green line represents the estimated value without GPU acceleration.

We can roughly see that there is no effect on the accuracy of the estimated camera movement trajectory whether GPU acceleration is enabled or not. The absolute error between the trajectory estimated by the visual SLAM algorithm without GPU acceleration and the actual value is shown in Table 8. The absolute error between the trajectory estimated by the visual SLAM algorithm with GPU acceleration and the actual value is shown in Table 9.

Among them, the root mean square error (RMSE) is the square root of the ratio of the sum of squares of deviations

between the observations and the actual values to the number of observations. It is used to measure the deviation between the observed value and the actual value. STD (Standard Deviation) is the arithmetic square root of variance. It is used to measure the degree of dispersion of a set of numbers.

According to the data in Tables 8 and 9, we draw Figure 10.

From Figure 10, we can see that the camera motion trajectory estimated by the GPU acceleration visual SLAM algorithm and the camera motion trajectory estimated by the non-GPU acceleration visual SLAM algorithm are compared with the actual value; there is a slight difference in accuracy. But the algorithm execution efficiency of the former is better.

4. Conclusions and Future Work

Visual SLAM needs to process a large number of image data, so the performance requirements of computing hardware are relatively high, which limits the application of visual SLAM on embedded platforms. In this paper, we studied the front end of visual SLAM based on the embedded platform, and then, we proposed the front-end parallelization method. Finally, the visual SLAM system was implemented on the embedded platform through GPU parallelization, and the effectiveness of the whole system was verified through the datasets. The method we propose effectively improves the execution efficiency of the visual SLAM front-end algorithm on a high-performance embedded platform and can be comparable to better-performance laptop computers.

The visual SLAM is a huge and complex project. Due to time constraints, we have not done enough research on it. The visual SLAM based on embedded GPU studied in this paper can be further explored from the following three aspects:

(1) In this paper, the visual SLAM algorithm ORB-SLAM2, which is usually applied to desktop computers, is applied to an embedded computer with relatively limited computing resources. By using the embedded GPU on the embedded development platform to accelerate the visual front-end processing of this visual SLAM algorithm in parallel, our purpose is to improve the processing efficiency and reduce the time overhead of the algorithm in the embedded computer. Therefore, for the operation of the visual SLAM algorithm on the embedded development platform Jetson TX2, we only compared the computational efficiency of the algorithm with the CPU alone and with the GPU. This verifies the effectiveness and feasibility of the parallel computing approach proposed in this paper. In future work, we will compare the parallel computing method proposed in this paper with better parallel computing methods that are already available, so as to further improve our proposed parallel method

(2) With the advancement of technology, the performance of embedded and other miniaturized mobile platforms will become more powerful, such as higher -performance GPU or high-performance FPGA. These hardware devices can make visual SLAM algorithms more efficient

(3) When the camera moves too fast, the image texture information collected by the camera is not rich enough, the scene illumination changes drastically, and the visual SLAM will have large estimation errors or feature points loss. For this, an inertial measurement unit can be used for the multisensor fusion to compensate for the disadvantages of the visual sensor

Data Availability

The video datasets used to support the findings of this study can be downloaded from the public website address which is provided in the paper (https://projects.asl.ethz.ch/datasets/doku.php?id=kmavvisualinertialdatasets#ground-truth).

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the Sichuan Science and Technology Program (2019ZDZX0007 and 2019YFG0399).

References

[1] R. C. Smith and P. Cheeseman, "On the representation and estimation of spatial uncertainty," *The International Journal of Robotics Research*, vol. 5, no. 4, pp. 56–68, 1986.

[2] X. L. Huang and S. Y. Yu, "Image segmentation based on normalized cut and CUDA parallel implementation," in *5th IET International Conference on Wireless, Mobile and Multimedia Networks (ICWMMN 2013)*, pp. 209–214, Beijing, China, 2013.

[3] C. Y. Du and J. L. Yuan, "Real-time splicing of panoramic video with GPU acceleration and L-ORB feature extraction," *Computer Research and Development*, vol. 54, no. 6, pp. 1316–1325, 2017.

[4] C. Wu, S. Agarwal, B. Curless, and S. M. Seitz, "Multicore bundle adjustment," in *Computer Vision and Pattern Recognition (CVPR 2011)*, pp. 3057–3064, Colorado Springs, CO, USA, 2011.

[5] D. Rodriguez-Losada, P. San Segundo, M. Hernando, P. de la Puente, and A. Valero-Gomez, "GPU-mapping: robotic map building with graphical multiprocessors," *IEEE Robotics & Automation Magazine*, vol. 20, no. 2, pp. 40–51, 2013.

[6] R. Mur-Artal and J. D. Tardos, "ORB-SLAM2: an open-source SLAM system for monocular, stereo, and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[7] F. Fraundorfer and D. Scaramuzza, "Visual odometry : part II: matching, robustness, optimization, and applications," *IEEE Robotics & Automation Magazine*, vol. 19, no. 2, pp. 78–90, 2012.

[8] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[9] H. Bay, T. Tuytelaars, and L. V. Gool, "SURF: speeded up robust features," in *European Conference on Computer Vision (ECCV 2006)*, pp. 404–417, Berlin, Heidelberg, 2006.

[10] P. F. Alcantarilla, J. Nuevo, and A. Bartoli, "Fast explicit diffusion for accelerated features in nonlinear scale spaces," in *British Machine Vision Conference (BMVC 2013)*, pp. 13.1–13. 11, London, U.K., 2013.

[11] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: an efficient alternative to SIFT or SURF," in *International Conference on Computer Vision (ICCV 2011)*, pp. 2564–2571, Barcelona, Spain, 2011.

[12] Jetson TX2December 2020, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2.

[13] M. Larabel, "Benchmarks of many ARM boards from the Raspberry Pi to NVIDIA Jetson TX2," 2017, https://www.phoronix.com/scan.php?page=article&item=march-2017-arm&num=1.

[14] Corner DetectionAugust 2020, http://en.wikipedia.org/wiki/Corner_detection/.

[15] ASL DatasetsAugust 2020, https://projects.asl.ethz.ch/datasets/doku.php?id=kmavvisualinertialdatasets.

[16] M. Burri, J. Nikolic, P. Gohl et al., "The EuRoC micro aerial vehicle datasets," *International Journal of Robotics Research*, vol. 35, no. 10, pp. 1157–1163, 2016.

[17] M. Grupp, "Python package for the evaluation of odometry and SLAM," December 2020, http://github.com/MichaelGrupp/evo.