







Research and Application of Visual SLAM Based on Embedded GPU

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

Chengdu University of Information Technology, Chengdu, Sichuan, China
{wanglt,wut}@cuit.edu.cn

Abstract. In automatic navigation robots, 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 color, 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. In this regard, this paper uses embedded hardware equipped with embedded GPU, combines CUDA-based GPU parallel computing and visual SLAM algorithm, finally, designs a parallelization scheme based on embedded GPU.

Keywords: Visual-SLAM · Embedded · Parallel computing · CUDA · GPU

1 Introduction

1.1 Background

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 [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), Virtual Reality (VR), etc. 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 color 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 real-time operation on embedded platforms and mobile platforms a great challenge. 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.

1.2 Main Research Content

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. XianLou [2] uses CUDA to accelerate the processing of image segmentation algorithms based on normalization, and Chengyao [3] uses CUDA to optimize image feature extraction and realizes the real-time stitching of panoramic video, which overcomes the shortcomings of high power consumption, non-real-time and low stability that used to rely on post-processing.

The current development of visual SLAM has been relatively mature, and there are various types of solutions, including sparse method, semi-dense 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 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, such as Wu C [4] and Rodriguez Losada [5] have implemented beam adjustment and ICP (Iterative Closest Point) algorithms on GPU, but these works are all performed on desktop GPUs.

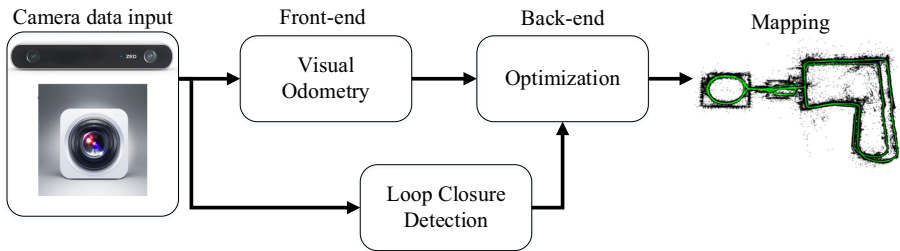


Fig. 1. A block diagram of a typical visual SLAM system.

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 system performs feature extraction and matching on these input images at the front end, and then roughly estimates the position of the feature points and the robot, and 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 transfers the optimized result to the closed-loop inspection to eliminate the accumulated error of the robot moving for a long time, and then uses 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, 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.

Finally, the parallel mechanism is analyzed 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 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 data set. The results show that the whole system is in good working condition. In addition, by counting the time overhead of executing data set, 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) [6]. 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: SIFT [7], SURF [8], ORB, AKAZE [9]. For details, please check the relevant literature.

2.2 ORB Feature Detection Algorithm

ORB algorithm was proposed by Ethan Rublee [10] and others 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. However, as the manufacturing technology gradually approached its limit, the growth rate of CPU computing performance has gradually slowed down. Even if CPU manufacturers represented by Intel and AMD have introduced multi-core 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, multi-threaded, multi-core processor architecture like today, with huge computing power and extremely high memory bandwidth. Its computing performance is far stronger than the CPU (see Fig. 2).

The reason behind this huge computing performance gap is that the difference in hardware structure between GPU and CPU. Let's start this topic from the "core" perspective. 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. In other words, although each core of the CPU is extremely powerful in processing tasks, it has fewer cores and does not perform well in parallel computing. In contrast, although the computing power of each core of the GPU is not powerful, it has a large number of cores, which can handle multiple computing tasks at the same time, and is well competent for parallel computing.

The different hardware features of GPU and CPU determine their application scenarios. The CPU is the core of the computer's operation and control, and the GPU is mainly used for graphics and image processing. The form of the image presented in the computer is a matrix. Our processing of the image is actually to operate various matrices

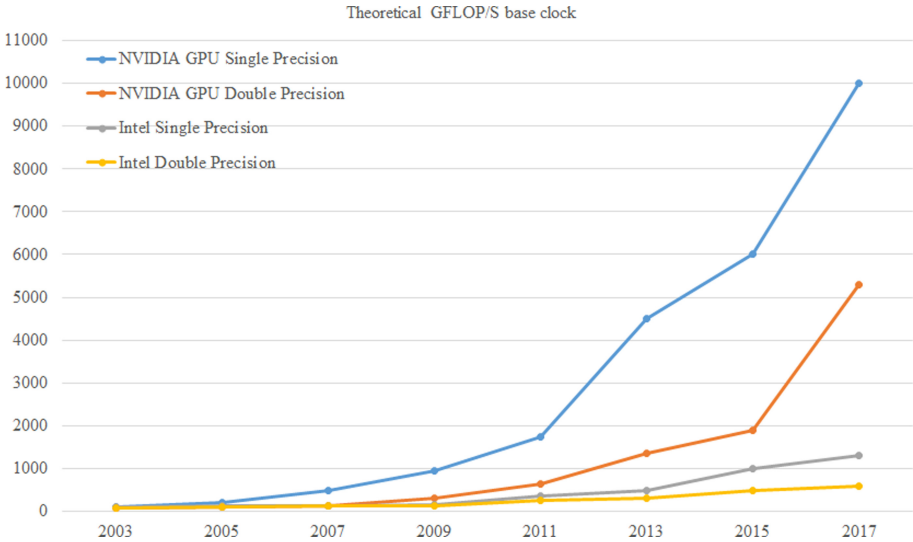


Fig. 2. Shows the intuitive comparison of single-precision and double-precision floating-point computing capabilities between CPU and GPU. It can be foreseen that the computing difference between CPU and GPU will become larger and larger in the short term.

for calculations, and many matrix operations can actually be parallelized, which makes image processing fast.

Now we compare the features of CPU and GPU from the perspective of data processing. The CPU needs strong versatility to handle a variety of different data types, such as integers, floating point numbers, etc., and it must be good at handling a large number of branch jumps and interrupt handling caused by logical judgments. So the CPU is actually a powerful processing unit. It can handle many things properly. Of course, we need to give it a lot of hardware resources for it to use, which makes the CPU impossible to have too many cores. The GPU is facing a highly unified, independent, large-scale data and a pure computing environment that does not need to be interrupted. Although the processing power of its core is far less powerful than that of the CPU, the GPU has a lot of cores, which makes up for the lack of single-core computing power and supports parallel computing.

It can be seen from the structure (see Fig. 3) 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



Fig. 3. Diagram of CPU and GPU structure.

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 developed rapidly in recent years.

In the current computer architecture, in order to complete CUDA parallel computing, the GPU alone cannot complete the computing task. The CPU must be used to cooperate to complete a high-performance parallel computing task. Generally speaking, the parallel code performs on the GPU and the serial code performs on the CPU. This is heterogeneous computing. Specifically, heterogeneous computing means that processors of different architectures cooperate with each other to complete computing tasks. The CPU is responsible for the overall program flow, and the GPU is responsible for the specific calculations. When each thread of the GPU completes the calculations, the results are copied to the CPU to complete a computing task (see Fig. 4).

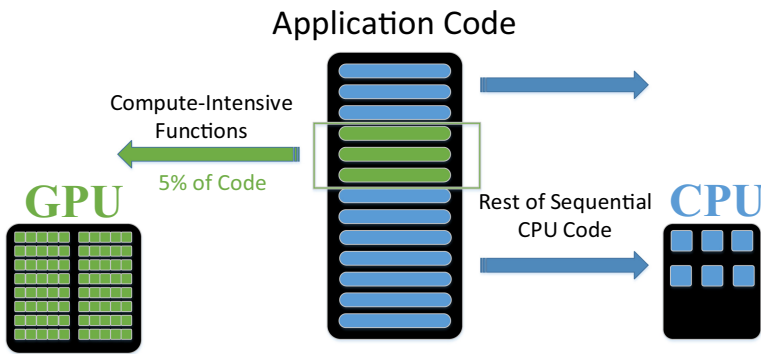


Fig. 4. The intensive calculation code (about 5% of the code amount) is completed by the GPU, and the remaining serial code is executed by the CPU.

3.2 CUDA Hardware Model

A simplified diagram of the GPU hardware architecture that supports CUDA is shown in Fig. 5. 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 Streaming Multiprocessor (SM), also known as 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.

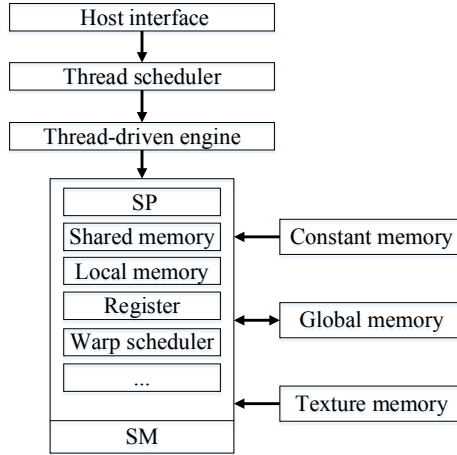


Fig. 5. GPU hardware architecture diagram.

The thread scheduler is responsible for allocating parallel tasks to SM. Each SM can start multiple thread blocks to execute parallel tasks, and the number of them is allocated by the developer according to their actual calculation needs. Each thread block is composed of multiple threads. Threads belonging to the same thread block can communicate and collaborate efficiently through shared memory. Finally, CUDA will allocate computing tasks to a certain number of threads, and these threads will eventually be mapped to each SP for calculation.

In addition, in order to meet the diverse needs of graphics rendering, the GPU has multiple types of memory for threads to access its stored data. These memories have their own characteristics. The characteristics of these memories are summarized in Table 1. Making good use of these memories can reduce unnecessary data access time and improve calculation speed.

3.3 CUDA Programming Model

The CUDA software environment is constantly updated with the development of the GPU hardware architecture. The latest version is CUDA 11.0. The functions provided by the entire environment are getting more and more powerful, and the interface becomes very friendly. CUDA supports C/C++, Python, JAVA and other high-level languages, and the corresponding programs are executed by the CPU and GPU. The CUDA parallel

Table 1. The characteristics of GPU memories.

Memory	Cache	Access permission	Life cycle
Register	×	Thread private	Thread
Local memory	×	Thread private	Thread
Shared memory	×	Shared within thread block	Block
Global memory	×	Device read/write	Grid
Constant memory	✓	Device read only	Grid
Texture memory	✓	Device read only	Grid

program executed on the GPU is also called a kernel function, which is specifically used to complete GPU parallel computing tasks, and its definition is also different from ordinary functions.

The following will introduce a typical CUDA program execution procedure (see Fig. 6). Each grid contains several thread blocks, which are composed of several threads. All thread blocks in the grid can be parallelized, and all threads in the thread block can also be parallelized, so the degree of parallelization is high.

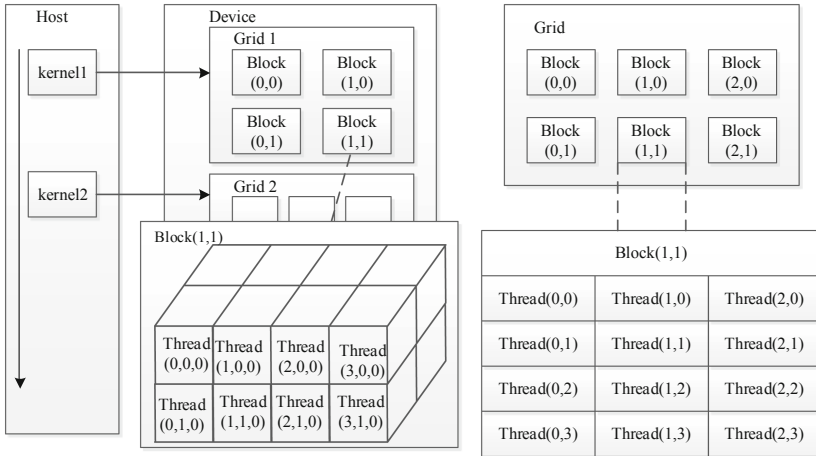


Fig. 6. Host refers to the host side, representing the CPU, and Device refers to the device side, representing the GPU. Grid is the outermost thread organization structure in CUDA.

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 Device to 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 multi-core 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. Its main performance indicators are shown in Table 2. The computing performance and related parameters of its GPU are shown in Table 3.

Table 2. TX2 main performance indicators.

Parameter	Performance indicator
CPU	ARM Cortex-A57(quad-core) 2 GHz + NVIDIA Denver2(dual-core) 2 GHz
GPU	256 CUDA-COREs Pascal 1300 MHz
Memory	8 GB 128-bit LPDDR4 1866 MHz 58.3 GB/s
Storage	32 GB eMMC
TDP	7.5–15w

Table 3. GPU related parameters of TX2.

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

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. In the following subsections, we will analyze 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.

Front-End Parallelizable Analysis. 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 key 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 key 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 non-maximum 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 feature extraction (see Fig. 7) is as follows:

1. Construct image Gaussian pyramid.
2. Perform FAST key points detection in each image layer of the pyramid.
3. Perform coordinate normalization in image layers of different sizes.
4. Delete duplicate FAST key points. Compare each key point with the corresponding key point in the upper and lower adjacent image layers at the same scale, and keep the key point with the largest response value.
5. Non-maximum suppression. Each key 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 key point is greater than the other key points, will it be kept, otherwise it will be deleted.
6. Sort key points according to FAST and Harris response values [11]. Select the top N best feature points, and the N value is preset according to requirements.
7. Assign the direction to each key point, and calculate the BRIEF descriptor to complete the extraction of ORB features.

Analyze the parallelization of the above steps, we can get the conclusion:

1. In the FAST key points detection procedure, there is no data communication between the image layers of the Gaussian pyramid, so FAST key point detection can be performed in parallel in each image layer.
2. FAST key points detection only has data association with each pixel and its neighboring pixels in the image, and the detection procedure is exactly the same, so it can be executed in parallel on a large scale.
3. Duplicate point deletion and non-maximum suppression are both related to the feature point and the image where it is located, and also related to the neighborhood feature points in the upper and lower adjacent images at the same scale. Therefore, the Gaussian pyramid of the three image layers can be input at the same time and calculated in parallel in the same way.

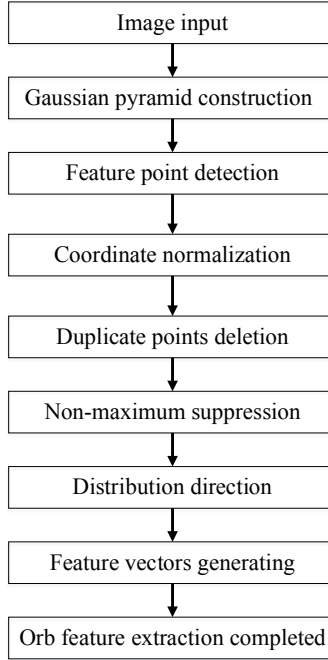


Fig. 7. ORB feature extraction flow chart.

4. For each key point, the calculation of its direction is based on the data association between the key point and the independent local image information in the image where it is located, so it can be calculated in parallel.

After the image feature extraction is completed, feature matching is needed to find the same feature point pair in the two images. There are many methods for feature matching. Due to the ORB feature vector (also known as feature descriptor) is a binary string, Hamming distance can be used to describe the similarity of a pair of feature vectors. After that, the same feature point pair can be found by the method of brute-force matching. Hamming distance can be expressed as:

$$D(V_1, V_2) = \sum_{i=0}^{255} x_i \oplus y_i \quad (1)$$

V_1, V_2 are two ORB feature vectors, $V_1 = x_0x_1 \cdots x_{255}$, $V_2 = y_0y_1 \cdots y_{255}$. The process of calculating the Hamming distance is to perform an exclusive OR operation on each bit of the two feature vectors. The smaller the value of $D(V_1, V_2)$, the higher the similarity of the two feature vectors; and the larger the value of x , the lower the similarity of the two feature vectors. In addition, a threshold K needs to be set. If $D(V_1, V_2)$ is greater than the threshold, the corresponding feature vector and feature point should be deleted. Finally, a rough matching point pair can be obtained by means of the method of brute-force matching.

With the matched point pairs, a change model can be established through the RANSAC (Random Sample Consensus) algorithm to describe the change relationship between the points in the two images:

$$[x_1, y_1, 1]^T = M[x, y, 1]^T \quad (2)$$

The transformation matrix M is written as:

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \quad (3)$$

Suppose P is the coordinate data set of the point pair obtained by rough matching, then select 3 pairs of matching points from P to calculate the transformation matrix M through Eq. 2. Then use the remaining point pairs in P to verify the accuracy of the transformation matrix M , and count the number of matching points that conform to the model. Repeat the above procedures, the final selected transformation model should have the most matching points.

Analyze the parallelization of the above steps, we can get the conclusion:

1. The calculation of the similarity of two ORB feature vectors has no data communication, so it can be processed in parallel.
2. When calculating the Hamming distance, the exclusive OR operation of each bit of the two feature vectors is only related to the binary value of the bit itself, so this procedure can be processed in parallel.
3. The RANSAC algorithm selects 3 pairs of points from the set P of matching points, which are independent for matching, and there is no data communication, so this procedure can be processed in parallel.
4. When calculating the transformation matrix M , it is only related to the matching points used, so this procedure can be processed in parallel.
5. Verify the accuracy of each transformation matrix M is only related to all rough matching point pairs, so this procedure can be processed in parallel.

The FAST Key Point Detection Parallelization Design. Map each image layer in the image pyramid to a thread, and each thread block allocates 32×8 threads, the number of thread blocks is:

$$N = \frac{W + blockDim.x - 7}{blockDim.x} \times \frac{H + blockDim.y - 7}{blockDim.y} \quad (4)$$

In Eq. 4, N represents the total number of thread blocks required to perform this task, $blockDim.x$ represents the number of threads in the x direction, $blockDim.y$ represents the number of threads in the y direction, W and H are the width and height of the image respectively. In order to prevent access conflicts in parallel threads, each thread in parallel will perform FAST key point detection on a pixel in the image. If it conforms the FAST key point, the coordinate of the pixel is saved in the global memory, and the total number of detected FAST key points is recorded and saved in the global memory.

In addition, a corresponding size of global storage space is allocated for each layer of image and initialized to zero to store the response value of key points, if a certain pixel is selected as the FAST key point, its response value is saved in the corresponding position of the allocated space.

Coordinate Normalization. In order to store the updated response value of the key point, the global memory is allocated the same size as the storage space of the original image, and each thread block is allocated 256 threads. The total number of thread blocks is the same as Eq. 4. Each thread in parallel calculates the normalized coordinate of a key point and updates the response value of the feature point after the normalized coordinate in the allocated space.

Duplicate Points Deletion and Non-maximum Suppression. The coordinate value and the response value of a feature point to be detected, the response value of the feature points in the neighborhood around the feature point to be detected, and the response value of the neighborhood feature points in the upper and lower scale image of the image layer where the feature point to be detected is located are transferred as parameters to the CUDA kernel function that performs duplicate points deletion and non-maximum suppression. And each thread block allocates 256 threads, the number of thread blocks is:

$$N = \frac{\overline{N}_{fast} + blockDim.x - 1}{blockDim.x} \quad (5)$$

In Eq. 5, N represents the total number of thread blocks required to perform this task, \overline{N}_{fast} is the total number of FAST key points of the image layer, each thread in parallel calculates one key point. Read the response value of all key points in the neighborhood of the key point in the three image layers. If the response value of the key point is the largest, keep this key point and delete the two key points at the corresponding positions in the upper and lower adjacent image layers at the same scale, otherwise delete itself. At the same time, the number of key points remaining after duplicate points deletion and non-maximum suppression operations should be recorded.

Filter Key Points. Use CUDA kernel function to sort the key points and corresponding response values remaining after the previous operation. Here directly use the parallel sorting algorithm in the Thrust library, and then according to requirement, select the top N feature points with the highest response value for subsequent calculations.

Distribution Direction. Calculate the direction of each feature point in parallel, and allocate 32×8 threads per thread block, the number of thread blocks is:

$$N = \frac{N_{fast} + blockDim.y - 1}{blockDim.y} \quad (6)$$

In Eq. 6, N represents the total number of thread blocks required to perform this task, N_{fast} is the total number of FAST key points. According to reference [12], each thread in parallel calculates the first-order moment m_{10} , m_{01} , and the origin moment m_{00} of the feature point. Then save the results in shared memory to improve the efficiency of

repeated access to data. In addition, thread 0 in each thread block is used to calculate the direction of the feature point according to reference [12], and the result is saved in the global memory after the calculation is completed.

Generating Feature Vectors. Transfer relevant parameters to the CUDA kernel function, and then start the kernel function to extract the ORB feature vector. And each thread block allocates 32×8 threads, the number of thread blocks is:

$$N = \frac{L + \text{blockDim}.x - 1}{\text{blockDim}.x} \times \frac{N_{\text{fast}} + \text{blockDim}.y - 1}{\text{blockDim}.y} \quad (7)$$

In Eq. 7, N_{fast} is the total number of FAST key points, and L is the length of the ORB feature vector. The feature vector is generated according to the direction of the key point and the sampling mode of the neighborhood of the image block in each thread block.

Feature Matching. Calculate the Hamming distance between two feature vectors in parallel by the method of brute-force matching. Each thread calculates the Hamming distance between a feature vector in the current image and all feature vectors in another image. Save the index value and Hamming distance value of the two feature vectors with the smallest distance in the global memory. Each thread block is allocated 256 threads, and the total number of thread blocks is:

$$N = \frac{N_{\text{vec}} + \text{blockDim}.x - 1}{\text{blockDim}.x} \quad (8)$$

In Eq. 8, N_{vec} is the total number of feature vectors in the current image. When the calculation is completed, the results saved in the global memory are transferred back to the memory on the Host side. Then filter out the mismatched results through a threshold to obtain rough matching results. When the rough matching is completed, all the saved data is transferred from the Host side memory to the Device side constant memory to provide data for the RANSAC algorithm.

RANSAC. First set the number of iterations K , and then randomly generate K groups of random numbers, each group contains three different random numbers. Since these random numbers will not be changed in subsequent calculations, these random numbers are transferred from the Host side memory to the Device side constant memory. Thereby, the data access speed can be improved through the cache in the constant memory.

Transfer relevant parameters to the CUDA kernel function, and then start the kernel function to calculate the transformation matrix M . Thread 0 in each thread block reads a set of random numbers from the constant memory to determine the three sample numbers. Then the feature point coordinates corresponding to the sample number can be read from the constant memory. Each thread in parallel uses the coordinates of the three feature points randomly selected to calculate the transformation matrix M according to Eq. 2 and save the result in the global memory.

Then transfer relevant parameters for the second CUDA kernel function, and start the kernel function to verify the accuracy of the transformation matrix M . Each thread in parallel reads the coordinates of the feature points from the constant memory to verify the accuracy of a transformation matrix M , and it is necessary to count the number of feature points that conform to the transformation matrix M . The best transformation matrix M is selected according to the number of feature points previously counted.

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 Dataset [13]. In the data sets, the data sets those name starts with MH are the videos shot indoors. Easy, medium, difficult represents the complexity of the scene in the video. As the complexity increases, high-speed moving scenes, scenes with strong lighting changes, and scenes with fast turning will appear in the video. These scenes will affect the accuracy of the estimated camera movement trajectory. During the experiment, the time cost of executing each set of data sets was recorded under the conditions of only using CPU and using GPU acceleration respectively. Time unit is minutes (min) (Table 4).

Table 4. The time overhead of executing data set

Data set	CPU only	CPU + GPU
MH_01_easy	7:51	4:56
MH_02_easy	6:31	3:53
MH_03_medium	5:41	3:31
MH_04_difficult	4:07	2:30
MH_05_difficult	4:46	2:47

From the results, the time cost of running the data set after using GPU parallelization on TX2 is greatly reduced, reducing the time by about half. Therefore, the use of GPU parallelization can significantly improve the processing performance of the visual SLAM front-end.

3.7 Movement Trajectory Estimation Test

Under the conditions of using only CPU and using GPU acceleration respectively, the camera motion trajectory is estimated through the visual SLAM algorithm. Due to the limited length of the paper, we only introduce the results of testing MH_04_difficult (see Fig. 8) and MH_05_difficult (see Fig. 9). In the two 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 whether GPU

acceleration is enabled has no effect on the accuracy of the estimated camera movement trajectory. This conclusion can also be obtained from Table 5 and Table 6.

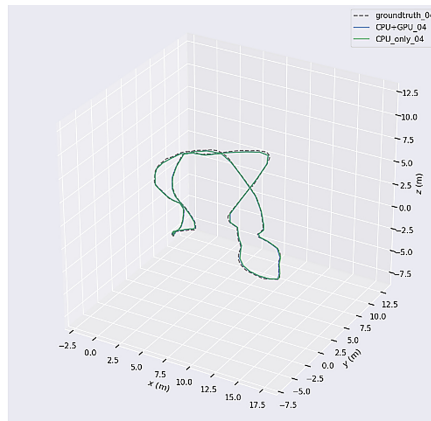


Fig. 8. Whether GPU acceleration is enabled has no effect on the accuracy of the estimated camera movement trajectory.

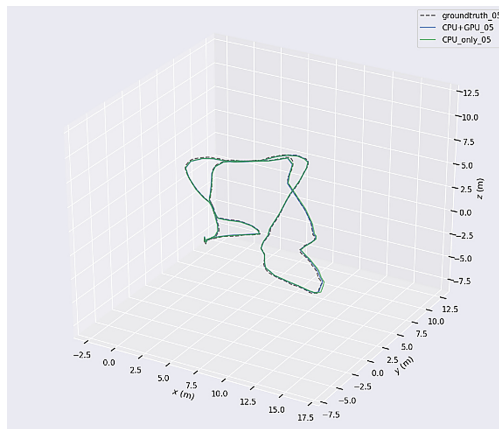


Fig. 9. Whether GPU acceleration is enabled has no effect on the accuracy of the estimated camera movement trajectory. However, at some corners of the trajectory, the result obtained by turning on GPU acceleration is closer to the actual value.

Table 5. In the MH_04_difficult, the absolute error between the two trajectory estimates and the groundtruth. The absolute error unit is meters (m).

Absolute error	CPU only	CPU + GPU
Max	1.132117	1.141322
Mean	0.424403	0.448092
Median	0.417360	0.439376
Min	0.028691	0.053778

Table 6. In the MH_05_difficult, the absolute error between the two trajectory estimates and the groundtruth. The absolute error unit is meters (m).

Absolute error	CPU only	CPU + GPU
Max	1.246482	1.224756
Mean	0.408896	0.421434
Median	0.387428	0.396226
Min	0.020689	0.048313

4 Conclusion 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 problem of visual SLAM based on embedded platform, and then we proposed front-end parallelization scheme. Finally, the visual SLAM system was implemented on the embedded platform through GPU parallelization, and the effectiveness of the system was verified through the data sets.

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 two aspects:

1. 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.
2. When the camera moves too fast, the image texture information collected by the camera is not rich enough, and the scene illumination changes drastically, the visual SLAM will have large estimation errors or data loss. For this, inertial measurement unit can be used for the multi-sensor fusion to compensate disadvantages of visual sensor.

Acknowledgement. This work was supported by the National Key Research and Development Program of China (No. 2018YFC1507005).

References

1. Smith, R.C., Cheeseman, P.: On the representation and estimation of spatial uncertainty. *Int. J. Robot. Res.* **5**, 56–68 (1986)
2. XianLou, H., ShuangYuan, Y.: Image segmentation based on Normalized Cut and CUDA parallel implementation. In: 5th IET International Conference on Wireless, Mobile and Multimedia Networks (ICWMMN 2013), Beijing, pp. 209–214 (2013)
3. Chengyao, D., Jinlin, Y.: Real-time splicing of panoramic video with GPU acceleration and L-ORB feature extraction. *Comput. Res. Dev.* **54**(6), 1316–1325 (2017)
4. Changchang, W., Agarwal, S.: Multicore bundle adjustment. In: *IEEE Computer Vision and Pattern Recognition*, pp. 3057–3064. Colorado Springs (2011)
5. Rodriguez, L.D., Segundo, P.S.: GPU-mapping: robotic map building with graphical multiprocessors. *IEEE Robot. Autom. Mag.* **20**(2), 40–51 (2013)
6. Fraundorfer, F., Scaramuzza, D.: Visual odometry: Part II: Matching, robustness, optimization, and applications. *IEEE Robot. Autom. Mag.* **19**(2), 78–90 (2012)
7. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vis.* **60**(2), 91–110 (2004)
8. Bay, H., Tuytelaars, T., Van Gool, L.: SURF: speeded up robust features. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) *ECCV 2006*. LNCS, vol. 3951, pp. 404–417. Springer, Heidelberg (2006). https://doi.org/10.1007/11744023_32
9. Alcantarilla, P.F., Nuevo, J.: Fast explicit diffusion for accelerated features in nonlinear scale spaces. In: *Proceedings of the British Machine Vision Conference* (2013)
10. Rublee, E., Rabaud, V.: ORB: an efficient alternative to SIFT or SURF. In: *IEEE International Conference on Computer Vision, Barcelona*, pp. 2564–2571 (2011)
11. Corner Detection. http://en.wikipedia.org/wiki/Corner_detection/. Accessed 10 Aug 2020
12. Xiang, G., Tao, Z.: *Fourteen Lectures on Visual SLAM: From Theory to Practice*. 2nd. Publishing House of Electronics Industry, Beijing (2019)
13. ASL Datasets. <https://projects.asl.ethz.ch/datasets/doku.php?id=knavvisualinertialdatasets>. Accessed 10 Aug 2020