2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP) | 978-1-6654-8308-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ASAP54787.2022.00018

Acceleration of video stabilization using embedded GPU

Yuzuki Mimura
University of Tsukuba
School of Science and Engineering
Tsukuba, JAPAN
yuzuki.mimura@darwin.esys.tsukuba.ac.jp

Chang Qiong
Tokyo Institute of Technology
School of Computing
Tokyo, JAPAN
cq@darwin.esys.tsukuba.ac.jp

Tsutomu Maruyama
University of Tsukuba
Intelligent and Mechanical Interaction Systems
Tsukuba, JAPAN
maruyama@darwin.esys.tsukuba.ac.jp

***Abstract*—Video stabilization is a technique used to eliminate the shakiness in video. It plays an important role to improve the quality of the videos captured by cameras mounted on drones and autonomous robots, and handheld cameras. In these cases, it is generally required to achieve real-time video stabilization using low-power and low-cost devices. In this research, we focus on software video stabilization, and propose a new implementation method using an embedded GPU. Our target device is Nvidia Jetson Nano, which is one of the smallest and least power consumption embedded GPUs. In our implementation, 1) multi-threads on the CPU on Jetson Nano and the GPU run asynchronously and in parallel to achieve high performance, 2) the size of the search area is minimized to reduce the amount of computation assuming that sufficiently fast processing speed is possible using the GPU, and 3) the data transfer from the global memory of the GPU is minimized to hide the high latency. Our implementation achieves 81.4 fps for full HD videos, and its quality is high enough for practical use.**

***Index Terms*—video stabilization, embedded GPU, real-time**

I. Introduction

Cameras have become smaller and less expensive as technology advanced, and are now used in various situations. In particular, it has become possible to mount high resolution cameras on drones and autonomous robots. However, videos captured by these cameras have undesired movement caused by the camera shake, and this movement makes it difficult to track objects of interest or extract their details from the scene. Video stabilization is a technique to eliminate high-frequency motion of the camera caused by shaking. Many approaches for video stabilization have been proposed [1]. They can be roughly divided into two categories: optical image stabilization (OIS) and electrical image stabilization (EIS). OIS is a hardware solution that uses a micro-electromechanical system gyroscope to detect the rapid camera movement, and its lens or image sensor is moved to eliminate the effect of the movement. EIS is a software solution that moves each frame of video in response to the movement. Another popular approach for video stabilization is to use a gimbal. However, its weight and cost may be an issue for drones and autonomous robots.

In this research, we focus on EIS using an embedded GPU. EIS allows the use of any camera, but its computational load is too large for micro-processors, and GPU acceleration is required for real-time processing. In our system, Nvidia Jetson Nano, which is one of the smallest and least power consumption GPUs, is used. Ths GPU is very inexpensive and can be used in various situations. Jetson Nano is a system on Module that includes a quad-core ARM A57 CPU, a 128-core Maxwell GPU, and 4 GB main memory. Thus, our system consists of only a camera and Jetson Nano.

In our system. the feature points in two consecutive frames are detected using FAST (Features from Accelerated Segment Test) [2], and their feature vectors are generated using BRIEF (Binary Robust Independent Elementary Features) [3]. The feature vectors are, then, compared to find the matching pairs, and an affine transformation matrix with feedback rate is generated from the matched pairs, and applied to the current frame to cancel the undesirable movement of the camera. This processing sequence is the combination of well-known methods, but each method is chosen and optimized to achieve high performance using a small embedded GPU.

The main features of our implementation using the GPU are as follows. First, different tasks are assigned to the CPU and GPU, and they run asynchronously and in parallel. Second, tasks that are executed continuously on the GPU are combined in the same kernel to mask the overhead caused by task switching. At the same time, the task and data assignment to each thread on the GPU are carefully designed to hide high memory access latency to the global memory of GPU. Finally, the size of search area of the matching, which often becomes the bottleneck of the computation, is minimized under the assumption that high speed processing is possible with this search area size reduction and the above two approaches. If the processing speed (fps) can be doubled, the distance of the movement caused by the camera shake between two consecutive frames becomes half, and the size of search area of the matching points can be reduced to 1/4. This means that the required computation speed can be reduced to half by doubling the frame rate.

This paper is organized as follows. The related works are described in Section II, and Nvidia Jetson Nano is introduced in Section III. In Section IV, the details of our implementation are given, and Section V shows how parallel processing is performed in our implementation. The experimental results are shown and discussed in Section VI. Section VII concludes the research, and discusses future work.

2160-052X/22/$31.00 ©2022 IEEE
DOI 10.1109/ASAP54787.2022.00018

TABLE I
SPECIFICATION OF JETSON NANO

CPU	Quad-core ARM A57 @ 918 - 1479 MHz
GPU cores	Maxwell 128 core @ 640 - 921.6 MHz
memory	4 GB 64-bit LPDDR4 25.6 GB/s
video decode	4K@60 |2x4K@30 |8x1080p@30 |18x720p@30 [H.264/H.265]
power mode	5W / 10W

II. PREVIOUS WORKS

In this section, we focus on GPU-based software video stabilization. Only a few studies exist on real-time software video stabilization for HD images using GPUs.

In [4], S. Aldegheri et al. performed video stabilization on an autonomous surface vehicle using an embedded GPU, Nvidia Jetson TX1 which consists of a quad-core A57 CPU, a 256-core Maxwell GPU, and 16 GB main memory. In this research, the Harris Response and FAST [2] are used to extract feature points in images, and optical flow is used to determine the camera movement. OpenVX is used for its implementation on the GPU, and a processing speed of 60 fps is achieved for full HD videos (1920×1080).

In [5], Dávid Pacura et al. proposed a GPU implementation method for video stabilization. In this implementation, each frame is divided into eight sub-areas, and the camera movement is determined using local binary patterns. A GPU for desktop computer with 384 cores, Nvidia 560 Ti, is used, and 31.7 fps is achieved for full HD videos.

III. EMBEDDED GPU JETSON NANO AND ITS PROGRAMMING

In our research, Nvidia Jetson Nano, an embedded GPU, is used. Table I shows the specification of this GPU board [6]. Jetson Nano is a system on Module, and it consists of a quad-core ARM A57 CPU, a 128-core Maxwell GPU, and 4 GB main memory. It also has support for video decoding and encoding. Unlike other GPU boards, Jetson Nano can run on its own, and Linux operating systems can be executed on the board alone. Ubuntu 18.04 is installed on the board. The power consumption of Jetson Nano is extremely low. It has two power modes, 5W/10W, and its true power consumption is lower. This board can be used on drones and autonomous robots because of its low-power consumption and its small size. However, Jetson Nano has only 128 GPU cores, and the acceleration by this board is limited.

The GPU architecture in Jetson Nano is nearly identical to that of other GPUs. The GPU consists of cores, shared memory and global memory. The global memory is large but slow. The shared memory is a fast memory similar to a cache memory, but its size is limited, and programmers must decide what data is to be stored in it. This data selection has a significant impact on performance.

The GPU in Jetson Nano is programmed using CUDA like other Nvidia's GPUs. In the programming by CUDA, thread, block and grid are used. A thread is the smallest unit of processing that is executed on a GPU core. A block is a set of threads. A shared memory is shared by the threads in the same block, and all threads in the same block become active at the same time. A grid is a set of blocks, and all threads in the same grid execute the same kernel.

IV. IMPLEMENTATION METHOD

In our implementation, camera movement is estimated by comparing two consecutive frames, the current frame F_t and the previous frame F'_{t-1}. Here, F'_{t-1} is the transformed one from F_{t-1} following the processing steps described below. The current frame F_t is, then, moved and rotated to counteract the estimated undesired camera movement, and F'_t is generated. The input to the system is the sequence of F, and the output is the sequence of F', which is the stabilized video.

The basic framework of our implementation is as follows.

A. F_t is read from a camera, and it is gray-scaled.
B. Feature points in the two frames, F_t and F'_{t-1}, are detected using FAST [2].
C. Their feature vectors are calculated based on BRIEF [3].
D. Feature point matching is performed between the two frames by comparing the feature vectors, and the matched pairs are obtained.
E. An affine transformation matrix M_t that transforms F_t to F'_{t-1} is generated using the matched pairs.
F. Another affine matrix with feedback rate M_t^* is constructed from the matrix to counteract the undesired movement.
G. M_t^* is applied to F_t, and F'_t is generated.

Step A, E and F are executed on CPU, and other steps are executed on GPU. By this task sharing, the processing of two frames, F_t and F_{t+1}, can be partially overlapped. These steps are for non-optimized implementation. In our actual implementation,

H. Feature points and the vectors of only F_t are calculated, and by comparing them with those of F_{t-1} that were stored in the global memory of GPU during the computation of F_{t-1}, M_t^* can be derived following the optimized method described in Subsection IV-H using M_{t-1} that was also stored in the global memory. Using this optimization method, the feature points and vectors of only F_t are calculated, and their computation time can be reduced by half.

Step A to H are described below in detail. The processing method is first described In each subsection, followed by the details of its implementation. These details are very important to achieve high-speed processing especially on GPU. The implementation is optimized for the processing of full HD images (1920×1080 pixels).

In the following discussion, GPU memory means the GPU global memory, and CPU memory referes to the CPU main memory.

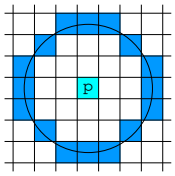

Fig. 1. The pixels on the circle used in FAST

A. Reading Frames from Camera and Preprocessing

One frame of an mp4 video stream is read from an external device, often camera, and decoded using a hardware decoder on Jetson Nano in pipeline processing. GStreamer is used to read the frame. The decoded frame is stored in the CPU memory. Since color information is not necessary to detect the feature points, this frame is copied and gray-scaled to simplify the calculations in the subsequent steps. Only 960×540 pixels in the center area of the frame are gray-scaled as described in the next subsection. GPU can be used to accelerate this gray-scaling, but considering the load balance between the CPU and GPU, the CPU is used in our current implementation.

This sequence is one of the most time consuming steps. The next frame is processed in parallel with the other steps of the current frame by using a new thread running on another core of the CPU.

B. Detecting feature points

FAST [2] is used to detect feature points. In FAST, feature points are detected as follows.

1) Two threshold values, t and N, are chosen.
2) Let p be the target pixel, and its brightness I_p.
3) The brightness of 16 pixels on the circle centered with p (Fig.1) is compared with I_p.
4) If more than N consecutive pixels satisfy $I_i < I_p - t$ (I_i is the brightness of a pixel on the circle), or $I_p + t < I_i$, p is considered a feature point.

In our GPU implementation, one pixel is assigned to one GPU thread, and parallel processing is performed. Block size is set to 128×1, and 128 pixels on the same line are processed simultaneously. In this case, 7×7 surrounding pixels for each pixel are required, three pixels each on the top, bottom, left, and right, and their total becomes a 128×7 pixel block as shown in Fig.2. These 128×7 pixels are copied into the fast shared memory in the GPU from the global memory because the values of these pixels are referenced multiple times. Fig. 2 shows three areas covered by three blocks, $i-1, i$ and $i+1$. Six pixels on each side of these areas are overlapped. This is because the detection cannot be performed correctly for three pixels at each end in 128×1 pixels, since the pixels on their left or right side are out of the 128×7 pixel block. Consequently, in each block, $128 - 3 \times 2 = 122$ pixels are tested if they are feature points.

In our current implementation, only the pixels in the center area of each frame (960×540 pixels) is used because of the following two reasons. First, lens distortion affects the peripheral part of the frame, and correct results may not be achieved, and second, the computation time of the following steps as well as this step can be reduced by limiting the number of feature points.

C. Calculation of the feature vectors

To compare the feature points. a feature vector, a vector representing its characteristics, is generated for each feature point. This vector is generated using BRIEF (Binary Robust Independent Elementary Features) [3]. BRIEF generates a binary vector of pixel p as follows.

1) Define a binary function to compare the brightness of two pixels u and v as follows.
   $f(u,v) = \begin{cases} 1 & if I_v \leq I_u \\ 0 & otherwise \end{cases}$
   where I_u and i_v are the brightness of u and v.
2) Consider a window consisting of 31×31 pixels centered with p.
3) Generate D pixel pairs (u_d, v_d) $d = 1, ..., D$ by randomly selecting $2 \times D$ pixels in the window.
4) For D pixel pairs, apply the function f, and generate a binary vector of length D.

A vector with more information can be generated by using larger D, but this requires more computation time. According to our experiments, $D = 128$ is sufficient.

In our GPU implementation, the same assignment of pixels to threads as the previous step are used. The kernel function of each thread can be summarized as follows.

```
FAST_and_BRIEF kernel() {
  index = blockDim.x * blockIdx.x + threadIdx.x;
  p is the pixel specified by index;
  if (p is not in the 960 × 540 pixel center area)
    return;
  if (p is a feature point) {
    calculate its BRIEF vector;
  }
}
```

The probability that a pixel will be a feature point is quite low, therefore most threads will be idle while BRIEF vectors are being calculated by other threads. However, data transfer from/to the global memory between FAST and BRIEF kernel becomes unnecessary with this implementation that continuously executes the feature point detection and the BRIEF vector calculation. According to our experiments, this simple approach is fast enough.

For calculating a BRIEF vector, 31×31 pixels around the target pixel p are required. In this case, they are not cached in the shared memory as is the case with feature point detection, and they are read from the global memory. This is because the probability that a pixel will be a feature point is extremely low, and their caching does not work efficiently.

The 128b in BRIEF vector is packed into two `long int`, and stored with its coordinate into the buffer allocated in the GPU memory using `atomicADD`.

D. Matching of BRIEF vectors

BRIEF vectors in two consecutive frames stored in the GPU memory are compared to find the matching of the feature

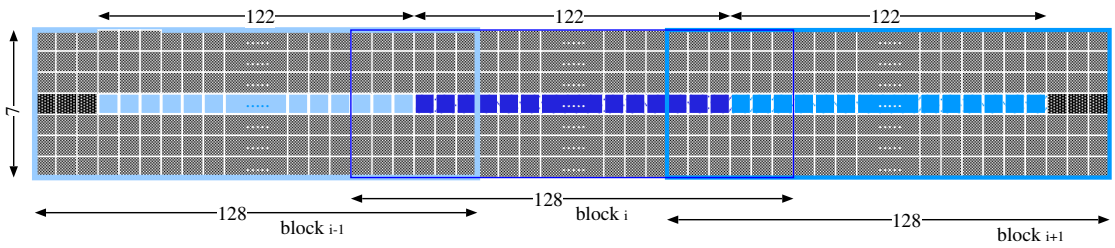

Fig. 2. Thread implementation of FAST

points in the two frames. In this matching, for each vector, the one with the minimum Hamming distance is searched. The calculation of Hamming distance between the two binary vectors can be calculated efficiently using bit operation instructions. In our system, each feature vector is compared with only those that are close enough in coordinates to reduce the number of calculations of Hamming distance. The processing speed of our system guarantees the upper limit of the moving distance of the feature points between two consecutive frames, and enables this search area reduction without losing the accuracy.

In our GPU implementation, one thread is assigned for each set of a feature vector and its coordinate in frame F_t, and each thread executes the following kernel. The parallelism in this matching step becomes the number of the feature vectors, namely the number of the feature points, in frame F_t. Here, the feature vectors and their coordinates in frame F_t are stored in a buffer `fbuf[0][]` in the GPU memory, and those of frame F'_{t-1} are in `fbuf[1][]`.

```
matching_kernel() {
  index = blockDim.x * blockIdx.x + threadIdx.x;
  p = fbuf[0][index]; // the index-th set in F_t
  m = the number of the sets in F'_{t-1};
  for (int i = 0; i < m; i++) {
    q = fbuf[1][i]; // the i-th set in F'_{t-1}
    if (q.coordinate is close enough to p.coordinate) {
      d = Hamming_distance(p.vector, q.vector);
      if (d < d_min) {
        d_min = d;
        qsmallest = q;
      }
    }
  }
  return qsmallest.coordinate;
}
```

This kernel will cause the inefficiency in computation due to warp divergence caused by skipping the vector comparison based on distance in coordinates, but according to our experiments, this approach is faster than when the skipping is not done.

In this implementation, the size of each block is 128×1, and $n/128$ blocks (n is the number of feature points in frame F_t) are used.

E. Affine Transformation Matrix

Using the matched pairs of the feature points in two frames F'_{t-1} and F_t, an affine transformation matrix M_t that shows the camera movement between the two frames is estimated. In this step, the least squares method and Random Sample Consensus (RANSAC) are used to make the estimation robust to noise. This step is executed on the CPU.

First, we show a method to estimate M_t from the matched pairs using the least squares method. Let the components of the matrix M_t be a, b, c, d, e, f, and the coordinates of one matched pair be (x_i, y_i) and (u_i, v_i). Then, the following equation holds.

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = M_t \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix}$$

From this equation, the values of a, b, c, d, e, f can be calculated so that two error functions for the x and y components, $\Phi_x = \sum(au_i + bv_i + c - x_i)^2$ and $\Phi_y = \sum(du_i + ev_i + f - y_i)^2$, are minimized. Their values are given as follows.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum u_i^2 & \sum u_i v_i & \sum u_i \\ \sum u_i v_i & \sum v_i^2 & \sum v_i \\ \sum u_i & \sum v_i & n \end{bmatrix}^{-1} \begin{bmatrix} \sum x_i u_i \\ \sum x_i v_i \\ \sum x_i \end{bmatrix}$$

$$\begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} \sum u_i^2 & \sum u_i v_i & \sum u_i \\ \sum u_i v_i & \sum v_i^2 & \sum v_i \\ \sum u_i & \sum v_i & n \end{bmatrix}^{-1} \begin{bmatrix} \sum y_i u_i \\ \sum y_i v_i \\ \sum y_i \end{bmatrix}$$

In our implementation, RANSAC, which is robust to outliers, is also used to estimate the matrix more accurately. The algorithm of RANSAC can be given as follows.

1) k pairs among n pairs (the total number of matched pairs) are selected randomly.
2) Using these k pairs, a matrix, C_t, is estimated using the least squares method.
3) Each of the unselected $n - k$ pairs is evaluated using C_t, and if its error is smaller than a given threshold, the pair is considered to be an inlier.
4) If the number of inliers is larger than a given threshold, a new matrix C'_t is calculated using the inliers and k pairs, and their mean squared error is calculated for them.
5) If the mean squared error is smaller than previous ones, C'_t is stored as the best matrix.
6) The above steps are repeated several times.

The outliers can be efficiently removed with this method, and the affine matrix can be more accurately estimated. In our current implementation, the library in OpenCV is used.

F. Transformation and Correction Process

The affine transformation matrix obtained in the previous step is the one that shows all movement between the two frames. If this matrix is fully applied, the intentional camera movement by videographer is also canceled as well as the unintentional movement. A feedback rate is introduced to relax the transformation by the affine matrix, This relaxation makes it possible to follow the intended movement of the camera. This step is also processed on the CPU.

Let M_t be the affine transformation matrix obtained in the previous step. The center point of F_t, (c_x, c_y), is transformed to (m_x, m_y) using this matrix as follows.

$$\begin{bmatrix} m_x \\ m_y \\ 1 \end{bmatrix} = M_t \begin{bmatrix} c_x \\ c_y \\ 1 \end{bmatrix}$$

Next, we consider two values; θ_o and and S_o, the rotation angle and the magnification rate by M_t. These values can be easily determined from the components of M_t. Let the feedback rate be r. Then,the rotation and magnification for applying the feedback, θ_r and S_r, are given as

$$\begin{aligned} \theta_r &= (1-r)\theta_o \\ S_r &= \frac{1}{S_o}. \end{aligned}$$

This feedback for the magnification means that no stabilization along the distance direction is performed because the size in F_t was scaled up by S_o and it is corrected by $1/S_o$. The effect of movement along the distance direction is extremely small except when taking close-up shots. With these values, the matrix for the feedback is given as follows

$$M_t^{fb} = \begin{bmatrix} \alpha & \beta & (1-\alpha)m_x - \beta m_y \\ -\beta & \alpha & \beta m_x - (1-\alpha)m_y \\ 0 & 0 & 1 \end{bmatrix}$$

where

$$\begin{aligned} \alpha &= S_r cos\theta_r \\ \beta &= S_r sin\theta_r. \end{aligned}$$

This matrix is the one for the rotation by θ_r around (m_x, m_y), and the magnification by S_r.

Then, the matrix M_t^* that is applied to the current frame F_t to eliminate the undesired rotation and magnification is given as

$$M_t^* = M_t^{fb} M_t.$$

Here, the feedback for the movement along the x and y axis is not yet considered. Suppose that the center point (c_x, c_y) is transformed to (m'_x, m'_y) by the matrix M_t^*.

$$\begin{bmatrix} m'_x \\ m'_y \\ 1 \end{bmatrix} = M_t^* \begin{bmatrix} c_x \\ c_y \\ 1 \end{bmatrix}$$

In M_t^*, the components that determine the movement along the x and y axis are $M_t^*[0][2]$ and $M_t^*[1][2]$ respectively. By rewriting these components as follows, the feedback for the movement along the x and y axis can be applied.

$$\begin{aligned} M_t^*[0][2] &= rm'_x + (1-r)c_y - m'_x \\ M_t^*[1][2] &= rm'_y + (1-r)c_y - m'_y \end{aligned}$$

Finally, by applying M_t^* to F_t, F'_t is obtained as follows

$$F'_t = M_t^* F_t$$

and, F'_t is used as the stabilized frame of F_t.

G. Applying the Affine Matrix

For applying the affine matrix M_t^* to frame F_t using the GPU, OpenCV CUDA function `cv::cuda::warpAffine` is used.

H. Optimized Computation Method

Next, we introduce an efficient method for computing the affine transformation matrix. In the basic computation method, when a new frame F_t is given, the feature points in F_t are detected, and their feature vectors are calculated. At the same time, those in F'_{t-1}, the transformed frame from F_{t-1}, are also detected and calculated. Using them, the affine matrix M_t and its feedback matrix M_t^{fb} are calculated, and, F'_t is obtained as follows.

$$F'_{t-1} = M_t F_t \quad (1)$$

$$F'_t = M_t^{fb} M_t F_t \quad (2)$$

Here, the description of these equations is not correct because the matrices in these equations accept the coordinate of a pixel and transform it to another coordinate, but this description is used in this discussion because of its simplicity. Then, the same calculation sequence is applied to F'_t and F_{t+1}, and F'_{t+1} is obtained as follows.

$$F'_t = M_{t+1} F_{t+1} \quad (3)$$

$$F'_{t+1} = M_{t+1}^{fb} M_{t+1} F_{t+1} \quad (4)$$

With this basic method, it is necessary to detect feature points and calculate their feature vectors in both F'_t and F_{t+1}.

In the optimized method, when F_{t+1} is given, the feature points and their feature vectors in only F_{t+1} are calculated. Then, using those of F_t that have already been detected and calculated, M'_{t+1}, an affine transformation matrix from F_{t+1} to F_t, is calculated.

$$F_t = M'_{t+1} F_{t+1}$$

Using this equation, Equation (2) can be rewritten as follows.

$$F'_t = M_t^{fb} M_t M'_{t+1} F_{t+1} \quad (5)$$

From Equation (3) and (4), the following equation can be derived.

$$F'_{t+1} = M_{t+1}^{fb} F'_t$$

Then, by assigning Equation (5) to this equation,

$$F'_{t+1} = M_{t+1}^{fb} F'_t = M_{t+1}^{fb} M_t^{fb} M_t M'_{t+1} F_{t+1}$$

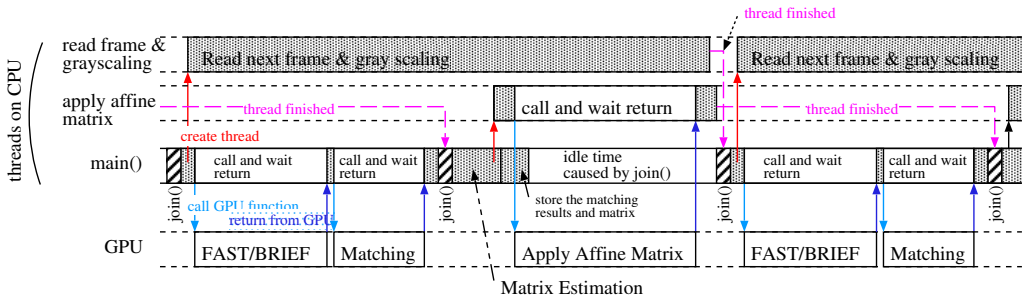

Fig. 3. Parallel Processing Using CPU and GPU on Jetson Nano

can be obtained. By comparing this equation with Equation (4),

$$M_{t+1} = M_t^{fb} M_t M'_{t+1}.$$

can be obtained. Since $M_t^{fb} M_t$ was already calculated during the calculation of F'_t, we can obtain M_{t+1} from M'_{t+1}. M_{t+1}^{fb} can be calculated directly from M_{t+1} as described in the previous subsection.

With this optimization method, it becomes possible to obtain F'_{t+1} without detecting feature points in F'_t and calculating their feature vectors. Feature point detection and their vector calculation are one of the most time consuming steps in our implementation, and this optimization technique is very important to achieve high processing speed.

V. Parallel Processing by CPU and GPU

The steps that are processed by the GPU can be significantly accelerated with the implementation described above. But, all steps cannot be accelerated by the GPU, and the processing time of the steps executed on the CPU becomes longer than that of the GPU. Among them, the step for reading and decoding the next frame from the camera is the most time-consuming one, and its processing time can reach the total processing time on the GPU.

Therefore, it is necessary to overlap these steps with the processing by the GPU. The CPU in Jetson Nano has four cores. In our implementation, three threads are executed in parallel on this CPU. The outline of the codes is shown below.

```
1  int main() {
2    int rd_id, mx_id = -1;
3    thread_create(&rd_id, read_frame);
4    for (;;) {
5      thread_join(rd_id);
6      thread_create(&rd_id, read_frame);
7      FAST_and_BRIEF();          // executed on GPU
8      Matching();                // executed on GPU
9      if (mx_id > 0) thread_join(mx_id);
10     estimate_affine_matrix();
11     thread_create(&mx_id, apply_affine_matrix);
12     store_matching_results_and_matrix();
13   }
14 }
15
16 void read_frame() { //read one frame from camera
17   read_frame_from_mp4_stream_and_decode();
18   gray-scaling();
19 }
20
21 void apply_affine_matrix(){
22   cudamemcpy(PC to GPU); // color frame
23   cv::cuda::warpAffine();  // executed on GPU
24   cudamemcpy(GPU to PC);
25 }
```

Three threads, `main()`, `read_frame()` and `apply_affine_matrix()`, are executed in parallel, and the GPU is called asynchronously from `main()` and `apply_affine_matrix()`.

In `main()`, first, the first frame in the mp4 stream is read and decoded by creating a new thread that executes `read_frame()` (line 3). After waiting for the thread to finish (line 5), a new thread is immediately created to read the next frame (line 6). Then, `FAST/BRIEF` and `Matching` for the read frame are performed using the GPU (line 7 & 8). If the thread for applying the affine matrix to the previous frame has already been terminated (line 9), the affine matrix for the current frame is estimated (line 10), and applied to the current frame using a new thread (line11). Finally, the matching results and the estimated matrix are stored in the CPU memory for the optimization method described in Subsection IV-H.

In `read_frame()`, one frame in the mp4 stream is read, decoded, and gray-scaled using one thread because there is no parallelism in this sequence. In `apply_affine_matrix()`, the color frame stored in CPU memory in `read_frame()` is copied to GPU memory, the affine matrix is applied by the GPU, and the result is copied back to the CPU memory.

The output of `apply_affine_matrix()` is the final result, and it should be transferred from Jetson Nano to the external device. This can be done using another thread. Although the CPU has four cores, only three of them are used.

Fig.3 shows how the three threads on the CPU and the calculation on the GPU are executed. This figure is the expected one because the three threads are running on Ubuntu 18.04, and their true running status cannot be known. In our system, as described in the next section, the processing time of the thread for `read_frame()`, and the total processing time by the GPU is comparable, and the other tasks are overlapped with them.

VI. Experimental Results

The performance of our implementation is evaluated from the two perspectives: its processing speed and the quality of the stabilized video.

Table II summarizes the comparison of processing speed with previous works described in Section II. As shown in

TABLE II
THE COMPARISON OF PROCESSING SPEED FOR FULL HD VIDEOS

	CPU	GPU	#cores	fps
Aldegheri et al. [4]	ARM A57	Jetson TX1	256	60
Pacura et al [5]	-	Nvidia 560 Ti	384	31.7
our system	ARM A57	Jetson Nano	128	81.4

TABLE III
THE COMPARISON OF PROCESSING TIME ON CPU AND GPU (μSEC)

step		CPU ARM A57	CPU + GPU ARM A57+128 cores (speedup)
A	read one frame & gray scaling	15146	11032 / -
B,C	feature points detection & vector calculation	36245	- / 2468 (14.7)
D	matching	7579	- / 717 (10.6)
E,F	estimate matrix	813	709 / -
G	applying the affine matrix	41247	- / 2319 (17.8)
	other steps	2620	8384 / -
	time to process one frame (throughput)	47257 21.2 fps	12278 81.4 fps

this table, our system achieves a higher processing speed with fewer hardware resources. The power consumption cannot be known, but considering the device used in each system, our one is considered to be the lowest. The processing speed of our system is fast enough for a wide range of applications.

Table III compares the processing time of each step described in Section IV for full HD 1920×1080 pixel images by only the CPU on Jetson Nano, and by the CPU and GPU. The maximum frequency of the CPU, ARM A57, is 1478 MHz, and that of 128 GPU cores is 921.6 MHz as shown in Table I. The numbers in this table are the average of 300 frames. Each number in Table III is not the time spent on the CPU or GPU, but the difference in time between when the task is started and when it is completed including any idle state caused by any reason. The execution time on the GPU includes the time spent transferring data from the CPU to GPU before the GPU processing, and that from GPU to CPU after the processing. The time was measured using `std::chrono`.

Step A, step E,F, and other steps are executed on only the CPU in both cases. The values in these lines should be the same, but actually, they are considerably different. For the processing on the CPU, three threads are used, and the execution of each thread can be suspended depending on the state of the other threads. The difference in the processing time of the other steps is the largest. It mainly consists of the time spent storing the matching results and matrix as shown in Fig.3, and no other tasks wait for it to complete. It seems that the considerable differences are caused by this interaction between the three threads.

When only the CPU is used, the time to process one frame (the throughput processing time) is 47,257 μsec (21.2 fps). This fast processing speed is achieved because of the fully optimized calculation method in our system. In this case, steps B to F are executed sequentially on one thread because of their strong data dependency. The sum of the processing time of these steps is 44637 $(36245 + 7579 + 813)$, and is very close to the throughput processing time. The most time consuming step is step G, and its processing time, 41247, is also close to the throughput processing time. These mean that, step A, step B to F, and step G are efficiently executed in parallel on the three threads. The acceleration rate by using the three threads is 2.19 because the sum of all execution time is 103,650 μsec.

The throughput processing time when the GPU is used is 12,278 μsec (81.4 fps), which is fast enough for various applications. Step B,C, step D and step G are accelerated using the GPU. Their acceleration rates are given in the parentheses right to the processing time, and they are 10.6 to 17.8. The acceleration rate for step D, 10.6, is the lowest because of the warp divergence caused by skipping the vector comparison as described in Subsection IV-D. That for step G, 17.8, is the highest because no warp divergence occurs throughout its calculation. The total processing time on the GPU is 5,312 $(2,468+715+23,19)$, and this is about half of the throughput processing time. Step A, which takes 11,032 μsec, dominates the throughput processing time. Its breakdown is 1) 5,331 μsec for copying the mp4 data of one frame from outside and to decode it using hardware decoder on Jetson Nano in pipeline processing, 2) 3,421 for copying the decoded data, and 3) 2280 for the gray-scaling. We also evaluated the processing time when the gray-scaling was performed on the GPU, but the total processing time could not be improved.

It seems that our throughput processing time is very close to the upper limit by Jetson Nano, because our processing time is very close to the processing time of step A, and it is difficult to reduce this processing time anymore because of the strong data dependency in it.

To evaluate the quality of the stabilized video, the rotation angle, and the movement along the x and y axis of frames in the original and stabilized videos are compared. It is desirable to compare these values to those by previous works, but for this comparison, their source programs and datasets are required, and it is very difficult in reality. We used the dataset in [7] for this evaluation. In this evaluation, the feedback rate r is fixed at 0.95. The original and stabilized video by our system can be seen in [8]. Fig.4 shows the rotation angle and the movements of 300 consecutive frames in the original and stabilized video. The rapid changes in these graphs are thought to have been caused by the camera shake. As shown in these graphs, the changes in the stabilized video are very smooth, and this means that the undesired rapid rotation and movements caused by the camera shake are well eliminated.

PSNR (Peak signal-to-noise ratio) has been used in several previous works, but we did not use it because it does not always offer the current evaluation results when it is calculated for frames with movement without the correct answer.

Fig.5 shows five sets of frames in the 300 frames used for the graphs in Fig.4. The five sets are #110, #120, #130,

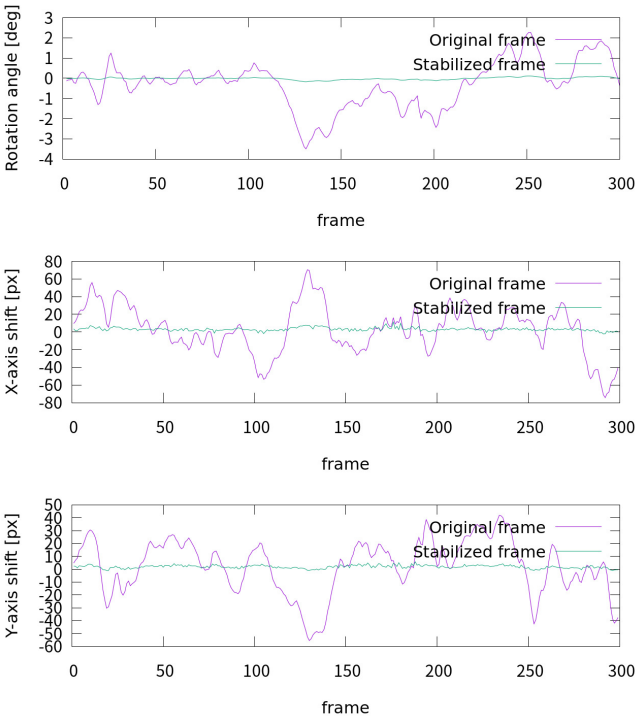

Fig. 4. Rotation and Movements of the original and stabilized videos

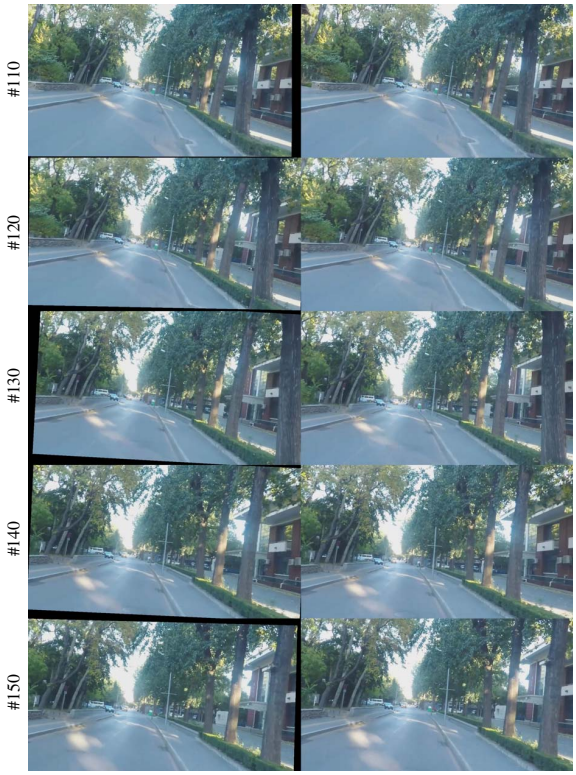

Fig. 5. Frame 110, 120, 130, 140 and 150 in stabilized and original video

#140 and #150. Five frames in the left column are from the stabilized video, while those in the right column are from the original video. In frame #110, as shown in Fig.4, the rotation angle is almost zero, the shift along the x axis is large, and that along the y axis is small. In Fig.5, the stabilized frame, the left one in the first row, is almost not rotated, but moved considerably to the left, and moved slightly upwards to cancel the movement. By the rotation and movements, some areas of the original frame are rotated and moved out of the frame, and black parts are brought into the frame. For the visibility, the periphery of these frames can be cropped. In the same way, in frame #130, as shown in Fig.4, the rotation angle is large as well as the shift along the x and y axis. In Fig.5, the stabilized frame, the left one in the third row, is rotated and moved considerably to cancel the rotation and movements. The large rotation in the original frame can be seen from the tilt of a tree at the right end that is almost vertical unlike the ones in other frames.

VII. Conclusions and Future Work

In this paper, we proposed a new implementation method of video stabilization using an embedded GPU, Jetson Nano, which is one of the most smallest and least power consumption GPUs. In our method, the amount of calculation is reduced aiming at processing speed faster than 60 fps, and the task assignment to the CPU and GPU is carefully designed to reduce the data transfer overhead. By using parallel processing of multi threads on the CPU cores and the GPU cores, we achieved 81.4 fps for full HD videos of 1920×1080 pixels. This processing speed is fast enough for various applications.

To further improve the implementation method and to achieve real-time processing on 4K video is the main future work.

References

[1] R. Marcos e., H. d. A. Maia, and H. Pedrini, "Survey on digital video stabilization: Concepts, methods, and challenges," *ACM Computing Surveys*, vol. 55, pp. 1–37, 2022.

[2] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Computer Vision – ECCV*, 2006, pp. 430–443.

[3] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *Computer Vision – ECCV*, 2010, pp. 778–792.

[4] S. Aldegheri, D. Bloisi, J. Blum, N. Bombieri, and A. Farinelli, *Fast and Power-Efficient Embedded Software Implementation of Digital Image Stabilization for Low-Cost Autonomous Boats*, 01 2018, pp. 129–144. [Online]. Available: https://www.researchgate.net/publication/320837423_Fast_and_Power-Efficient_Embedded_Software_Implementation_of_Digital_Image_Stabilization_for_Low-Cost_Autonomous_Boats/

[5] D. Pacura and M. Drahanský, "Cuda accelerated real-time digital image stabilization in a video stream," *International Journal of Software Engineering and its Applications*, vol. 10, pp. 113–124, 2016.

[6] "Power management for jetson nano and jetson tx1 devices." [Online]. Available: https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/power_management_nano.html

[7] M. Wang, G.-Y. Yang, J.-K. Lin, S.-H. Zhang, A. Shamir, S.-P. Lu, and S.-M. Hu, "Deep online video stabilization with multi-grid warping transformation learning," *IEEE Transactions on Image Processing*, vol. 28, no. 5, pp. 2283–2292, 2019.

[8] [Online]. Available: https://drive.google.com/drive/folders/1JN_rWk1QEjyFY6ZzO5qTFfw-bDfvEZqc