BFROST: Binary Features from Robust Orientation Segment Tests accelerated on the GPU

Jaco Cronje
Council for Scientific and Industrial Research, Pretoria, South Africa
Email: jcronje@csir.co.za

***Abstract*—We propose a fast local image feature detector and descriptor that is implementable on the GPU. Our method is the first GPU implementation of the popular FAST detector. A simple but novel method of feature orientation estimation which can be calculated in constant time is proposed. The robustness and reliability of our orientation estimation is validated against rotation invariant descriptors such as SIFT and SURF. Furthermore, we propose a binary feature descriptor which is robust to noise, scalable, rotation invariant, fast to compute in parallel and maintains low memory consumption. The proposed method demonstrates good robustness and very fast computation times, making it usable in real-time applications.**

***Index Terms*—Computer vision, Feature detection, Feature extraction, FAST, GPU, BFROST**

I. Introduction

Feature detection forms an important part of many computer vision algorithms. Online image processing algorithms need real-time performance, thus the speed at which features are detected is crucial in many applications. Applications such as Visual SLAM (Simultaneous localization and mapping), image registration, 3D reconstruction and video stabilization need to match corresponding image features between multiple views. The detected corners or feature points need to be described unambiguously so that the correspondence between multiple views can be computed reliably. Real-time processing requires the feature detection, description and matching to be as fast as possible.

Modern day commodity GPUs (Graphics Processing Units) provide a way to implement general purpose parallel algorithms. The CUDA (Compute Unified Device Architecture) [1] framework from NVidia provides a programmable interface for GPUs.

FAST (Features from Accelerated Segment Tests) [2], [3] is one of the fastest and most reliable corner detectors implemented on the CPU (Central Processing Unit). By exploiting the parallel computation power of CUDA, we propose a solution to implement a similar feature detector on the GPU. To our knowledge a feature detector based on accelerated segment tests has not been implemented on the GPU before. The algorithm provides an additional orientation estimation for the detected feature that the original FAST implementation does not provide.

This paper also proposes a feature descriptor similar to recent features descriptors such as BRIEF (Binary Robust Independent Elementary Features) [4] and BRISK (Binary Robust Invariant Scalable Keypoints) [5], but which is robust to noise, maintains low memory consumption and is fast to compute, invariant to rotation, scale and lighting variations. The binary feature descriptor is fully implemented on the GPU with fast feature matching capabilities. We further refer to the proposed detector and descriptor as BFROST (Binary Features from Robust Orientation Segment Tests).

The remained of the paper is structured as follows: Feature detectors and descriptors in the literature related to our work is discussed in Section II. Our method is discussed in section III which contains a subsection III-A that describes the GPU implementation of our feature detector. Thereafter subsection III-B will provide information on our binary descriptor followed by section IV that demonstrates some comparable results. Finally section V concludes the paper.

II. Related Work

Many feature detector methods are available in the literature. Characteristics of robust detectors include invariance to image noise, scale, translation and rotation transformations. The well known SIFT (Scale-Invariant Feature Transform) [6] method is very robust, but the computation time is not feasible for real-time applications. GPU implementations for SIFT such as GPU-SIFT [7] have shown an improvement on computation time, but remain slow for hi-definition real-time video processing.

SURF (Speeded Up Robust Features) [8] improved on the computation time of SIFT by using an integral image for fast local gradient computations on an image. The open source computer vision library (OpenCV) [9] contains a GPU version of SURF. Although SURF executes faster than SIFT, SIFT remains more robust. Both SIFT and SURF describe a feature with a floating point vector. Matching between these vectors is usually performed by computing the squared distance between the vectors, which can be time consuming when matching thousands of features.

Recently, binary feature descriptors have received more attention. These descriptors are described with a binary string. The distance between two binary strings can be described by the Hamming distance. The Hamming distance can be rapidly computed by performing the bitwise XOR operator between the two strings and then calculating the number of set bits within the result. Modern day hardware architectures support instructions that can count the number of bits in a word rapidly. CUDA devices with compute capability 2.x maps the *__popc* instruction to a single hardware instruction that counts the number of set bits in a given word. Thus, matching binary descriptors can be computed rapidly.

BRIEF [4] computes pairwise pixel intensity comparisons to describe an image patch. Each comparison or test, results in a binary value that forms the binary string. The image patch surrounding the keypoint is initially smoothed before the 128, 256 or 512 comparison tests are then performed. The spatial locations of the pixels used in each test on the image patch are sampled from an isotropic Gaussian distribution. Figure 1 shows a typical BRIEF sampling pattern.

The BRIEF descriptor is fast to compute, the length of the descriptor is adjustable and matching can be performed efficiently. However, image patches need to be smoothed to be more robust against noise, the spatial sampling pattern needs to be rotated for the descriptor to be rotation invariant and the descriptor does not scale well because of the discrete pixel sampling.

BFROST scales well and is robust to noise because we use the intensity of regions for comparison tests instead of discrete pixels, our sampling pattern is not rotated to achieve rotation invariance.

BRISK [5] inspired by BRIEF, also computes pairwise comparisons to build the descriptor for an image patch. Instead of using randomly selected sampling points, the method uses a fixed sampling pattern consisting of 60 sampling points. Gaussian smoothing is applied on the patch of pixels surrounding each sampling point before the intensity value for each sample point is retrieved. Figure 2 shows the sampling pattern.

The sampling points of the sampling pattern are stored in a look-up table. This table contains a set of locations for each rotation which takes up a total of 40MB memory space. The rotation of the keypoint is estimated by examining the local gradients at each sampling point. The BRISK method also describes a method to determine the scale of the keypoint by using image pyramids and non-maximal suppression to search through the image scale-space.

BFROST does not require a large look-up table for our sampling pattern, we also don't need to perform Gaussian smoothing on the sampling patches, we use a very fast rotation estimation that requires no local gradient calculations.

The FAST [2], [3] detector detects keypoints by inspecting the pixel intensities of sixteen pixels on a circle surrounding the possible keypoint p. A positive classification occurs when there exist a set of n continuous pixels on the circle which are all brighter than the intensity $I(p) + t$ or all darker than the intensity $I(p) - t$, where t is the detection threshold value. Illustrated in Figure 3. The most repeatable results were obtained with $n = 9$. The strength of the keypoint is given by the maximal value of t that still classifies p as a keypoint, a simple binary search is performed to determine the maximum of t. Machine learning is applied to construct a decision tree classifier that can detect the feature at high speeds. The decision tree is converted into a set of nested if-else statements that can be compiled into C code.

FAST can quickly reject non-corner locations with only a few if statements. BFROST use the same continuous pixel-set criteria to detect keypoints. However using thousands of if-else statements on a GPU is not feasible, if even compilable. GPUs are very sensitive to branch instructions, especially if different branches executes within the same warp. The next section describes how we implemented the detector on the GPU without the need for thousands of if-else statements.

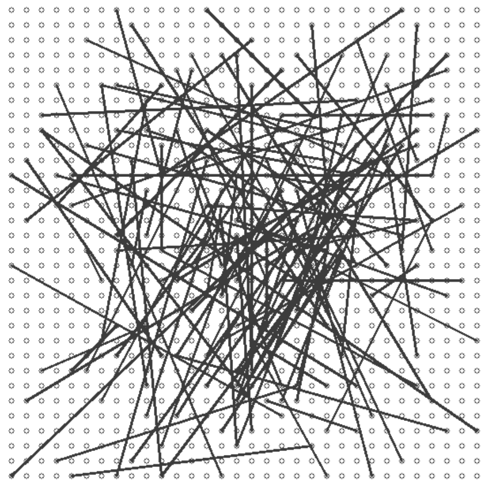
Fig. 1. Typical BRIEF sampling pattern. Each line indicates a pixel intensity comparison test. The image shows 128 tests. Taken from [4].

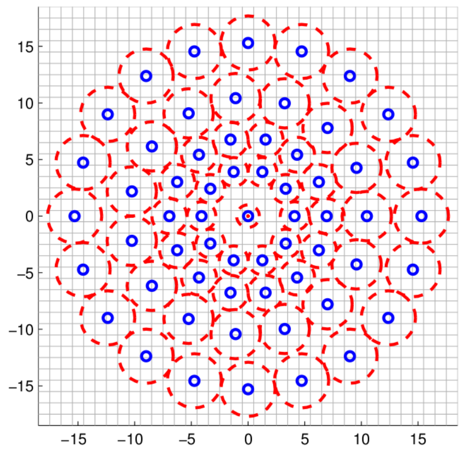

Fig. 2. The BRISK sampling pattern with 60 sampling points. The blue solid circles denote the sampling points and the red dashed lines indicates the standard deviation of the Gaussian smoothing applied at each sampling point. Taken from [5].

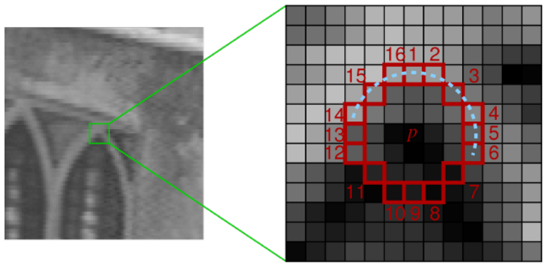

Fig. 3. The 9 point segment test corner detection, taken from [3]. The highlighted squares are the 16 pixels under inspection. The pixel at p is the possible keypoint. The dashed line is passing through 9 contiguous pixels which are all brighter than p by more than t.

III. BFROST: The Method

Inspired by the FAST [2], [3] detector, we desired a similar detector that is implementable on the GPU. A direct implementation of FAST in CUDA would not be feasible. Parallel execution performance dramatically decreases when different branches are executed within a block of threads on the GPU. The original detector only classifies a pixel as a corner or not and depends on other methods to extract the orientation information from the detected image patch. We propose a fast method for orientation estimation as part of our detector, described in Section III-A1.

Memory transfers between the CPU and GPU can become a bottleneck and should be kept to a minimum to achieve optimal performance. We keep the detected keypoint information on the GPU, perform non-maximal suppression and describe the keypoints with our binary descriptor without transferring excessive data between the GPU and the CPU.

Section III-A describes the feature detection method and Section III-B describes the feature description method, followed by a brief explanation on how to match binary descriptors efficiently in Section III-C.

A. Feature Detection

The first implementation optimization to note is the memory storage location of the input image on the GPU:

- We allocate a $CudaArray$ of the image size and bind the image data to this $CudaArray$. The $tex2D$ texture sampling instruction is used on the $CudaArray$, which is faster than reading from global memory. Texture sampling will cache the sampled data and increase the memory access performance.
- The texture filtering mode is set to point filtering such that discrete points are sampled. No interpolation is performed.
- The texture addressing mode is set to clamp any address that falls outside the size of the image.

Consider the bit string B formed by comparing the pixel intensity $I(p)$ of pixel p with the sixteen pixels on a circle around the pixel p under classification. B_i describes the bit value of the ith pixel comparison on the circle. Thus, i is in the range [0..15]. Let C_i be the ith pixel position on the circle and t the detection threshold.

$$B_i = \begin{cases} 1, & \text{for } I(p) + t < I(C_i) \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

The bit string B contains sixteen binary elements and when converted to a decimal value, falls in the range of [0..65535]. For each of these 2^{16} possible configurations, we determine if the configuration classifies p as a corner or not and store the binary result in a table. This look-up table of 2^{16} binary values are precomputed and stored as a binary string T. T is uploaded to constant memory on the GPU. Note that 2^{16} bits = 8192 bytes, which is less than the 64KB constant memory available on the GPU.

The feature detection process performs the following steps:

- Bind the input image to the $CudaArray$. Bind the look-up table T to constant memory (Only if the table has not been uploaded yet).
- Executes the feature detection CUDA kernel. One thread is assigned to each pixel in the image.
- Each thread loads the center pixel p and sixteen surrounding pixels C_{0-15} into local thread memory.
- Each thread builds the binary string B by performing (1) for all sixteen pixels on the circle.
- The binary classification value for B is read from T. If the image patch classifies as a keypoint, the (x, y) location of the pixel p is encoded into a 32 bit value and written into a 1-dimensional buffer containing all detected keypoints. The $AtomicInc$ CUDA instruction is used to retrieve the index at which the keypoint information should be written. The atomic increment operation needs to be used, because more than one thread can request to write keypoint information into the buffer at the same time.
- The binary string B and classification test is performed twice, once with the $I(p) + t < I(C_i)$ test and then with the $I(p) - t > I(C_i)$ test. The same look-up table T is used for both tests.

1) Rotation Estimation: The original FAST detector does not provide an orientation estimate. We propose a very simplistic method to estimate the rotation of a keypoint. Consider a keypoint detected with continuous brighter intensities from C_a to C_b where the detected segment length is $>= 9$. The rotation index τ and rotation angle θ can then be estimated by:

$$\tau = \begin{cases} \dfrac{a+b}{2}, & \text{for } a < b \\ \dfrac{a+b+16}{2} \mod 16, & \text{for } a >= b \end{cases} \tag{2}$$

$$\theta = \frac{2\pi\tau}{16} \tag{3}$$

We create a look-up table that maps the binary string B to the rotation index τ and upload the 2^{16} byte vector to the GPU for fast rotation estimation. The accuracy of the rotation estimation can be viewed in Section IV.

2) Non-maximal Suppression: The strength of each detected feature is calculated by finding the maximum value of t for which the image patch still classifies as a corner. The function on t is monotonic, which means that a simple binary search can be performed on t to quickly find the maximal value. Each thread in our CUDA kernel calculates the feature strength for one keypoint. A keypoint index map is created that maps an image location (x, y) to a keypoint index. The index map ensures that the eight neighboring keypoints of any given image location can be computed rapidly by performing an index look-up into the index map.

The non-maximal suppression kernel executes one thread for each keypoint. Each thread compares the feature strength of its own keypoint to the feature strengths of the eight neighboring keypoints. The index map is used to determine the index of the adjacent keypoints and whether they exist. All keypoints with a strength greater than all of the neighbors, gets written into the final detected keypoint buffer. The $AtomicInc$ atomic instruction is used again to avoid memory write conflicts.

B. Feature Description

Our fast binary feature descriptor relies on the usage of an integral image or summed area table to sample image intensity information from rectangular regions. Efficient GPU implementations for integral images [10] do exist. Similar to the BRIEF [4] and BRISK [5] descriptors, our descriptor creates a binary string by comparing the sum of intensities over regions within the image. The region based approach reduces the effect of noise and makes the descriptor scalable.

1) Sampling Pattern: The sampling locations of the BFROST sampling pattern stays fixed for all rotations and can be scaled accordingly. The base pattern is uploaded to constant GPU memory. Our pattern contains 64 sampling points, where the BRISK [5] pattern contains 60 sampling points and the number of BRIEF [4] sampling points equals the length of their descriptor. Our sampling point offsets $(X(i), Y(i))$ for each i in [0..63] with keypoint scale σ is calculated by:

$$r(i) = \sigma 2^{2+(i \mod 4)} \tag{4}$$

$$\phi(i) = \lceil \frac{i}{4} \rceil \tag{5}$$

$$X(i) = r(i) \cos(\frac{2\pi\phi(i)}{16}) \tag{6}$$

$$Y(i) = r(i) \sin(\frac{2\pi\phi(i)}{16}) \tag{7}$$

At each sampling point we sample the sum of all pixel intensities in a square region. Figure 4 shows the square regions associated with each sampling point. The width and height of the square region $Z(i)$ for each sampling point is determined by:

$$Z(i) = \frac{\pi r(i)}{8} \tag{8}$$

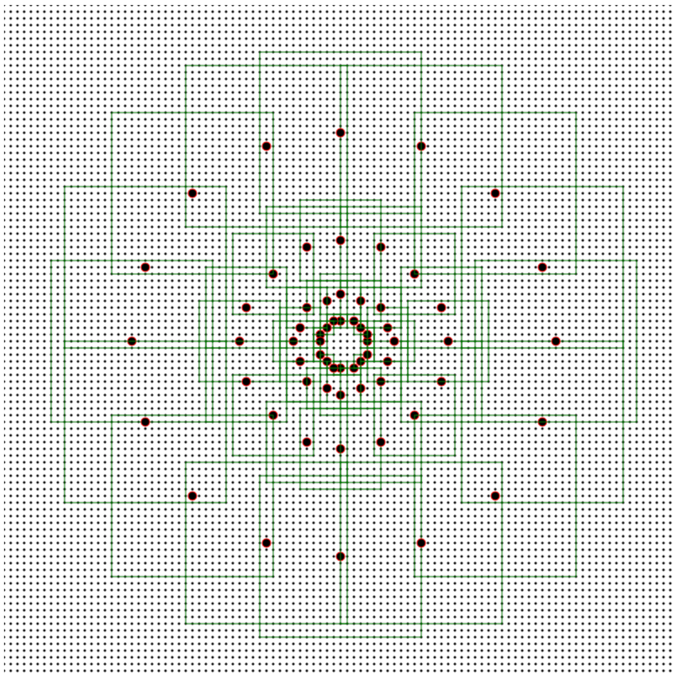

Fig. 4. 64 Sampling point locations with their associated square regions.

2) Building the Descriptor: The BFROST descriptor is 256 bits long. Each of the 64 samples is compared with 4 other samples to form the 256 bit descriptor. Let $S(i)$ be the average intensity of the square region sampled from sampling point i and let M be the integral image computed from the input image I. Let the location of the feature point j be $x(j)$ and $y(j)$ in image space. The area of the square region is given by $n(i)$.

$$n(i) = (2r(i) + 1)^2 \tag{9}$$

$$\begin{aligned} S(i) = (\quad & M(x(j) + X(i) + Z(i), y(j) + Y(i) + Z(i)) \\ & + M(x(j) + X(i) - Z(i), y(j) + Y(i) - Z(i)) \\ & - M(x(j) + X(i) + Z(i), y(j) + Y(i) - Z(i)) \\ & - M(x(j) + X(i) - Z(i), y(j) + Y(i) + Z(i))) \\ & \div n(i) \end{aligned} \tag{10}$$

The description process works as follows:

- The CUDA kernel creates 32 threads for each keypoint j. The block size is set to 512 threads. Thus, each block computes the descriptor for 16 keypoints.
- Each thread samples two square regions, $S(k)$ and $S(k+32)$ where k is the thread index for keypoint j. The normalized sampled results are stored in shared memory.
- A $syncthreads$ call is launched to ensure that all 32 threads receive the full set of 64 samples for the keypoint through shared memory accessible across threads within the same block.

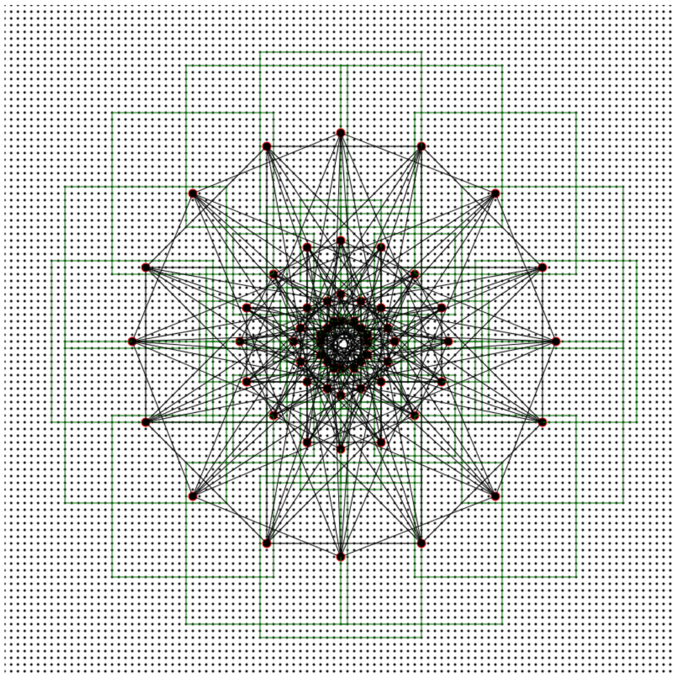

Fig. 5. Complete testing pattern, generating a 256 bit descriptor.

- Each thread computes 8 binary values by comparing the samples according to the sampling test pattern. The 8 bits are compacted into a byte and written into the descriptor global memory.

To form the 256 bit descriptor, each sample point i is compared with four other sample points. The index of these four sampling points is specified by $((i+8) \mod 64)$, $((i+24) \mod 64)$, $((i+36) \mod 64)$ and $(4\phi(i)+4+(3-(i \mod 4)))$. Figure 5 illustrates the complete testing pattern with 256 comparisons. To achieve rotation invariance on the descriptor, the index of the sampling points is simply modified by adding 4τ when performing the comparisons.

C. Fast Binary Feature Matching

The Hamming distance metric can be used to calculate the distance between two binary strings. Suppose we have two binary strings D_1 and D_2. Applying the XOR binary operator on D_1 and D_2 will result in a binary string D_{xor} that contains a binary 1 at each position where D_1 and D_2 differs. To calculate the Hamming distance, we only need to count the number of set bits in D_{xor}. CPU architectures support the *__popcount* instruction where as CUDA architectures support the *__popc* instruction. These instructions return the number of set bits within the given variable, which gives us the Hamming distance. Executing a XOR and *__popc* instruction is much more efficient than computing squared distances between floating point vectors.

Our current feature matching implementation performs a brute force matching scheme on the GPU. Future work will include faster descriptor matching algorithms such as BK-Trees [11]. Approximate string matching algorithms can be applied to the binary string matching problem that relates to the binary feature matching problem in Computer Vision.

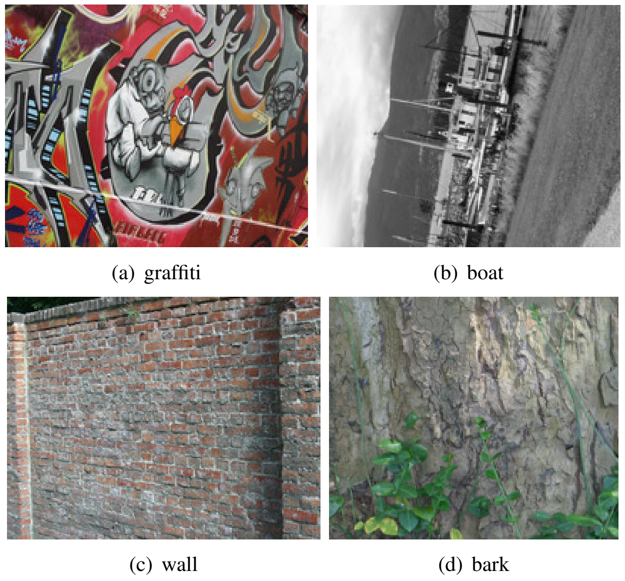

Fig. 6. Datasets used for evaluation.

IV. Results

The BFROST detector achieves similar repeatability when compared to the FAST detector. The GPU implementation executes faster as shown in Table I. The values reported are the averages over 100 runs. Our detector detects slightly more keypoints because the decision tree of FAST does not perform a complete segment test. Timing comparisons were performed on a NVidia GeForce GTX 460 for our GPU implementation and on a Intel Core i7 2.67 GHz for the OpenCV [9] FAST CPU implementation. Tests were performed on the first image in the graffiti, boat, wall and bark datasets shown in Figure 6. The detection threshold as defined in equation 1 was set to 40 for both detectors. Note that the time reported for BFROST includes feature detection, non-maximal suppression and orientation estimation while FAST only detected the features and performed non-maximal suppression.

TABLE I
Feature detection comparison.

Image	OpenCV FAST (ms)	Keypoints	BFROST (ms)	Keypoints
graf	5.271	996	0.676	1022
boat	9.971	5509	1.017	5725
wall	12.181	9696	1.229	9899
bark	2.790	312	0.544	320

For descriptor evaluation purposes we transform the test image with a known homography to produce a transformed image. Keypoints are detected and described on both images. For each keypoint in the test image we find the nearest neighbor in the transformed image based on the distance between the keypoint descriptors. We perform cross correlation by matching the keypoints in the transformed image with

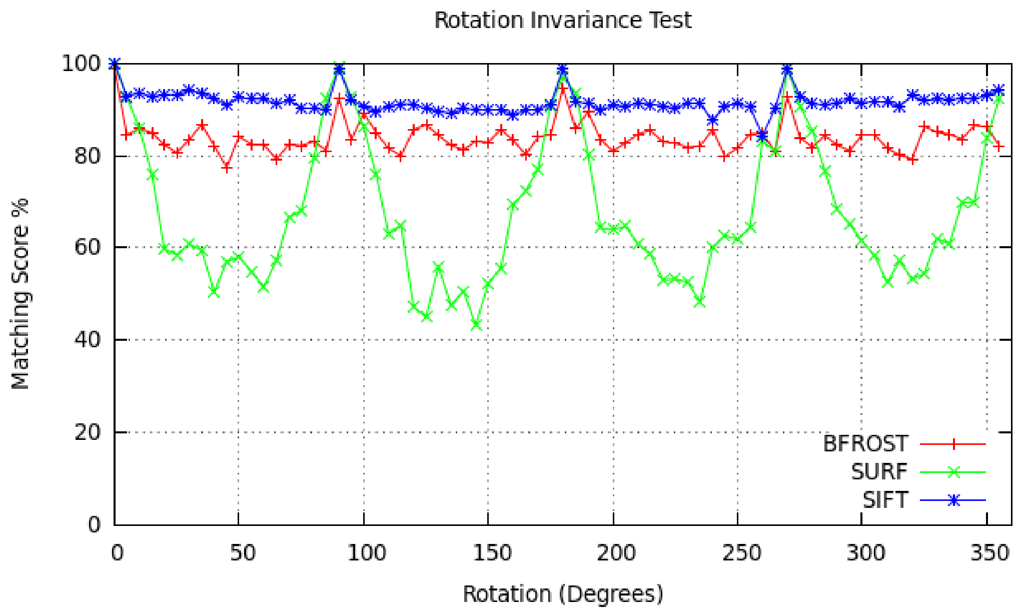

Fig. 7. Descriptor matching scores with in plane image rotations.

the keypoints in the test image and collect the matching keypoints that only matched both ways. Because we know the homography, the matching score is calculated by taking the ratio between the number of inliers and the total number of matched keypoints.

The graph in Figure 7 shows the rotation invariance of our detector and descriptor by applying in plane rotations to the graffiti test image. A higher matching score is better. We compare our results with SURF [8] and SIFT [6]. Table II shows the related computation time of each descriptor. Our detector and descriptor out performs the popular SURF [8] in rotation robustness and computation time. SIFT [6] remains more robust but falls far behind with computation time.

TABLE II

Feature description time comparison between descriptors with a test image rotated between 0 and 360 degrees with intervals of 5 degrees.

OpenCV SIFT	OpenCV SURF	BFROST descriptor
1259.98s	26.78s	0.08063ms

V. Conclusion

We have shown that a complete segment test detector can be implemented on the GPU and that the computation time of our detector is roughly 9 times faster than the popular fast FAST detector. An additional benefit to our approach is that the rotation estimation of the feature can be extracted with negligible extra computation cost. The robustness of our rotation estimation has been proven by comparison against SIFT and SURF.

We have proposed a fast binary feature descriptor, which is scalable, rotation invariant and more robust to noise than other binary descriptors such as BRIEF. The descriptor maintains a low memory requirement and doesn't use excessive look-up tables to obtain rotation invariance like BRISK. Fast binary descriptor matching can be performed resulting in BFROST being usable in real-time applications.

Future work will involve improving feature descriptor matching speed by implementing fast tree based matching algorithms on the GPU. Improving and incorporating accurate keypoint scale detection that improves on the BRISK method.

References

[1] NVIDIA-Corporation, "NVIDIA CUDA programming guide version 4.0."

[2] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking." in *IEEE International Conference on Computer Vision*, vol. 2, October 2005, pp. 1508–1511.

[3] ——, "Machine learning for high-speed corner detection," in *European Conference on Computer Vision*, vol. 1, May 2006, pp. 430–443.

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

[5] S. Leutenegger, M. Chli, and R. Siegwart, "BRISK: Binary robust invariant scalable keypoints."

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

[7] S. Sinha, J. Frahm, M. Pollefeys, and Y. Genc, "GPU-based video feature tracking and matching," in *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, vol. 278. Citeseer, 2006.

[8] H. Bay, T. Tuytelaars, and L. V. Gool, "SURF: Speeded up robust features," in *In ECCV*, 2006, pp. 404–417.

[9] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[10] B. Bilgic, B. Horn, and I. Masaki, "Efficient integral image computation on the GPU," in *Intelligent Vehicles Symposium (IV), 2010 IEEE*. IEEE, 2010, pp. 528–533.

[11] R. Baeza-Yates and G. Navarro, "Fast approximate string matching in a dictionary," in *String Processing and Information Retrieval: A South American Symposium, 1998. Proceedings*. IEEE, 1998, pp. 14–22.