Parallel ORB with OpenMP

Chenyu Wang,[*] Sid Wang,[†] Ziyue Feng,[‡] and Yihan Li[§]
New York University, New York, United States

The Oriented FAST and Rotated BRIEF (ORB) algorithm is a key player in feature detection and description within images. Our project focuses on the Rotated BRIEF part, which generates a string of 256-bit for each keypoint as its descriptor. It involves extracting 256 pairs of pixels surrounding the keypoint based on a designated pattern and making comparisons for each pair. To elevate computational efficiency. The adopted parallelization strategy aims to exploit multi-core processors to accelerate the descriptor computation phase, a critical step in image processing tasks. Our results demonstrate that applying OpenMP to parallelize the ORB algorithm is not only plausible but also scalable. Moreover, we observe enhanced performance as the size of keypoints increases.

Keywords: ORB, Parallel Computing, Image Processing

I. Introduction

ORB, combining Features from Accelerated Segment Test (FAST) for keypoint detection and Binary Robust Independent Elementary Features (BRIEF) for descriptor creation is first induced by Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R. Bradski in their 2011 paper[1]. It is widely utilized in computer vision, especially in pattern recognition and object detection. It integrates robustness against rotation and scale changes and thus outperforms the former methods. However, its computational intensity, particularly in BRIEF's descriptor computation, necessitates parallelization for improved performance.

II. Literature Survey

In the context of speeding up the ORB (Oriented FAST and Rotated BRIEF) descriptor, a common method involves leveraging the capabilities of OpenCV's CUDA-accelerated version.

The CUDA version of ORB in OpenCV, specified as cv::cuda::ORB, implements the ORB keypoint detector and descriptor extractor designed for GPU acceleration. This implementation enables certain optimizations such as the option to blur images before descriptor calculation, and functions like detect, computeAsync, and detectAndComputeAsync for efficient keypoint detection and descriptor computation on GPUs. An additional noteworthy development is the optimized GPU-accelerated feature extraction for ORB[6] This approach focuses on parallel GPU-based implementation for the keypoint extraction part. The key innovation here lies in accelerating the image pyramid construction on GPUs, which is a crucial component in ORB's functioning. However, there is limited information available on the use of multi-threading specifically for speeding up ORB descriptor extraction.

III. Proposed Idea

We propose the parallelization of the BRIEF descriptor computation within the ORB framework. This entails the distribution of pixel pair comparisons across multiple threads, thereby diminishing the overall computation time. OpenMP is employed for integrating parallelism into the ORB's existing structure. Additionally, we extend this parallelization strategy to the computation of descriptors for various keypoints, ensuring a comprehensive and efficient optimization of the ORB algorithm's performance.

```
function ORBDescriptor(keypoint, orientation):
    descriptor = []
    pixelPairs = DefinePixelPairs()

    rotationMatrix = CalculateRotationMatrix(
        orientation)
    rotatedPairs = RotatePixelPairs(pixelPairs,
        rotationMatrix)

    foreach pair in rotatedPairs:
        intensity1 = SampleIntensity(keypoint, pair
            [0])
        intensity2 = SampleIntensity(keypoint, pair
            [1])

        if intensity1 < intensity2:
            descriptor.append(1)
        else:
            descriptor.append(0)
    return descriptor
```

[*] Correspondence email address: cw4030@nyu.edu
[†] Correspondence email address: zw2686@nyu.edu
[‡] Correspondence email address: zf2182@nyu.edu
[§] Correspondence email address: yl10798@nyu.edu

IV. Experimental Setup

The purpose of the experiment is to compare the computation time of our multi-threaded ORB algorithm with two established benchmarks: the standard ORB implementation in OpenCV and the OpenCV CUDA-accelerated ORB.

In our experiments, we consistently use the same image (resolution: 4160*3120) across all tests for uniform evaluation. We vary the number of keypoints in this image to test the scalability and efficiency of our multi-threaded ORB algorithm under different computational loads. We also experiment with different thread counts in our algorithm to find the optimal balance between computation speed and accuracy. This helps us understand the trade-offs between processing speed and resource use, crucial for different computing environments.

To validate the correctness of our method, we conducted a thorough comparison between the outputs of our parallelized ORB implementation and the standard OpenCV ORB. This comparison aimed to ensure that our enhanced approach yields results consistent with the established OpenCV ORB output, thereby confirming the accuracy and reliability of our modifications.

A. Hardware setup

GPU server: cuda5.cims.nyu.edu

Component	Specification
CPU	2 Intel Xeon E5-2650 (2.60 GHz, 16 cores)
GPU	2 GeForce GTX TITAN Z (12 GB each)
System Memory	64 GB
OS	CentOS 7

B. Software setup

Software	Version
OpenCV	4.8.0
CMake	3.22
GCC	11.2
OpenMP	4.5
CUDA	11.4

V. Methodology and Approach

We present our modification of the main body of ORB descriptor computation. The original version (OpenCV) realization contains a nested for loop. The basic structure of the function is shown as pseudo code:

A. Original Version

```
function computeOrbDescriptors}(Keypoints, other per-
    keypoint info)
    for each k in keypoints:
        orientation = computeOrientation()
        pixelPairs = DefinePixelPairs()
        rotationMatrix = CalculateRotationMatrix(
            orientation)
        rotatedPairs = RotatePixelPairs(pixelPairs,
            rotationMatrix)
        for each pair in pixelPairs:
            compare pair and store bit into
                descriptor
        end
    end
end
```

B. Naive Implementation

A naive implementation of parallelism is to add "# pragma omp parallel for" before the outer loop.

```
function computeOrbDescriptors}(Keypoints, other per-
    keypoint info)
    # pragma omp parallel for
    for each k in keypoints:
        orientation = computeOrientation()
        pixelPairs = DefinePixelPairs()
        rotationMatrix = CalculateRotationMatrix(
            orientation)
        rotatedPairs = RotatePixelPairs(pixelPairs,
            rotationMatrix)
        for each pair in pixelPairs:
            compare pair and store bit into
                descriptor
        end
    end
end
```

C. Collapse the nested for loop

Although the above implementation realized parallelism, it is not efficient. We consider collapsing the nested for loop for further acceleration. Note that OpenMP only support collapsing a nested loop without any computation within the two loops. To this end, we put the per-keypoint computation into the inner for loop, where $\{\} \times \{\}$ denotes Cartesian production.

```
function computeOrbDescriptors}(Keypoints, other per-
    keypoint info)
    # pragma omp parallel for
    for each pair in {keypoints} X {pixelPairs}:
            orientation = computeOrientation()
            pixelPairs = DefinePixelPairs()
            rotationMatrix = CalculateRotationMatrix(
                orientation)
            rotatedPairs = RotatePixelPairs(
                pixelPairs, rotationMatrix)
            compare pair and store bit into
                descriptor
        end
    end
end
```

D. Pre-computation of per-keypoint data

We also observed that the computation of orientations and rotations of each keypoints is independent of pixel pair comparison. Thus, we pulled all per-keypoint computation out. This implementation is more efficient because now we compute per-keypoint data in contiguous arrays and thus it utilizes cache better.

```
function computeOrbDescriptors}(Keypoints, other per-
    keypoint info)
    # pragma omp parallel for
    for each k in keypoints:
        orientation = computeOrientation()
        pixelPairs = DefinePixelPairs()
        rotationMatrix = CalculateRotationMatrix(
            orientation)
        rotatedPairs = RotatePixelPairs(pixelPairs,
            rotationMatrix)
    end
    # pragma omp parallel for
    for each pair in {keypoints} X {pixelPairs}:
        compare pair and store bit into descriptor
    end

end
```

E. Avoid multiple creation of threads

In the above version, we create threads twice. To reduce this extra cost, we defined a global variable that stores the number of threads we want and create threads only once at the beginning.

```
function computeOrbDescriptors}(Keypoints, other per-
    keypoint info)
    # pragma omp parallel num_threads(
        global_num_threads)
    {
        # pragma omp for
        for each k in keypoints:
            orientation = computeOrientation()
            pixelPairs = DefinePixelPairs()
            rotationMatrix = CalculateRotationMatrix(
                orientation)
            rotatedPairs = RotatePixelPairs(
                pixelPairs, rotationMatrix)
        end
        # pragma omp for
        for each pair in {keypoints} X {pixelPairs}:
            compare pair and store bit into
                descriptor
        end
    }
end
```

Moreover, we further tested the setups includes different schedulers, load balancing options, and simd options. The experiment results shows that the default settings perform the best.

VI. Experiments & Analysis

The introduction of parallel computing into ORB led to a significant reduction in descriptor computation times across various test cases. The efficiency gains were more noticeable in larger images, indicating an effective harnessing of the multi-core architecture. Notably, the parallelized ORB maintained consistent feature detection and description quality compared to its standard counterpart.

a. Descriptor computation time : The total time that the descriptor-generation function executes.

b. Total computation time : The total time that ORB algorithm executes.

c. Total execution time : The total time that we execute the program, including reading the keypoints and writing descriptors to the txt files.

A. Experiment Results

Table I. Total computation time (ms) with different algorithms

# keypoints	OpenCV	CUDA	omp w/ 4 threads
1000	133.213	333.988	139.414
2000	184.971	311.198	145.496
5000	212.360	439.645	189.175
10000	216.040	364.329	188.070

B. Experiment Result Analysis

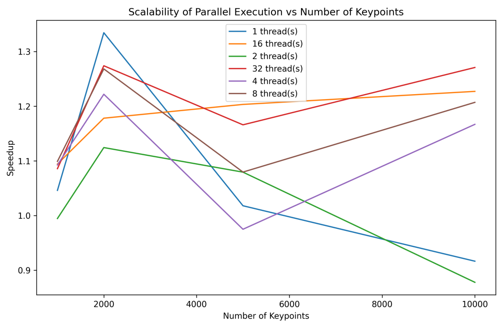
Table I highlights a performance comparison between the CUDA-based ORB algorithm and the CPU counterparts. It points out that the GPU implementation

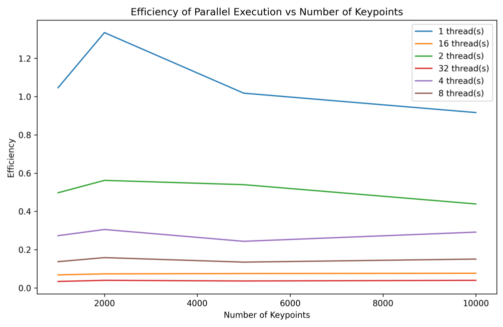
Table II. Descriptor computation time (ms) of parallel ORB with different number of threads

# kpts	OpenCV	1	2	4	8	16	32
1000	9.07	10.95	15.39	7.77	5.51	7.81	7.37
2000	21.93	21.75	29.14	16.03	8.66	10.04	8.38
5000	47.53	56.09	59.69	15.83	20.40	13.01	15.15
10000	89.49	112.06	107.60	62.90	34.15	17.64	21.66

of the ORB algorithm is notably slower than the CPU versions. This performance lag is attributed to the additional time required by the GPU to transfer data. We also observe that: as the number of keypoints increases, the advantages of the parallel ORB algorithm become more pronounced.

Table II and the images clearly demonstrate the scalability and efficiency of parallel ORB at different workloads and thread counts. The instances where a higher number of threads result in poorer performance than a lower number of threads could be attributed to the overhead created by the increased thread count. This overhead likely arises from the additional management and synchronization requirements associated with a larger number of threads, ultimately impacting overall efficiency.





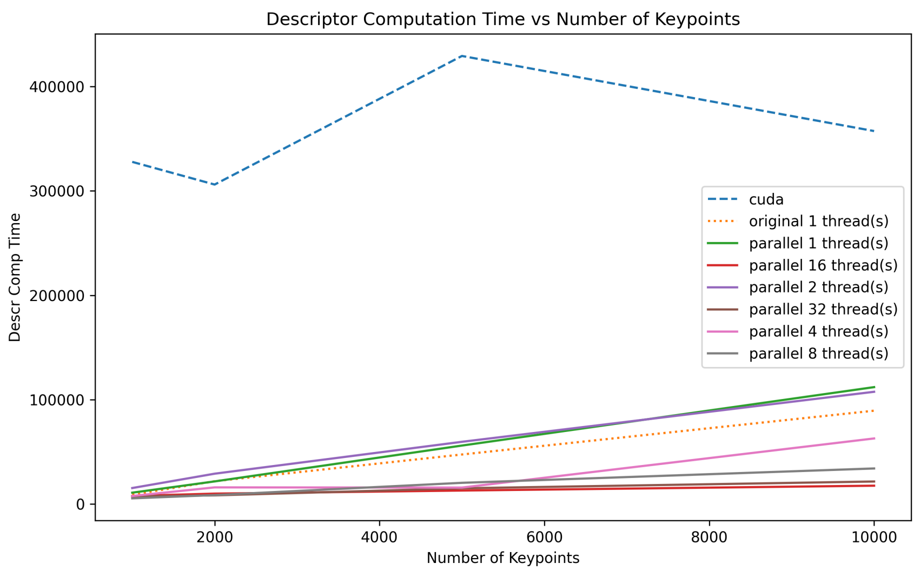
VII. Conclusions

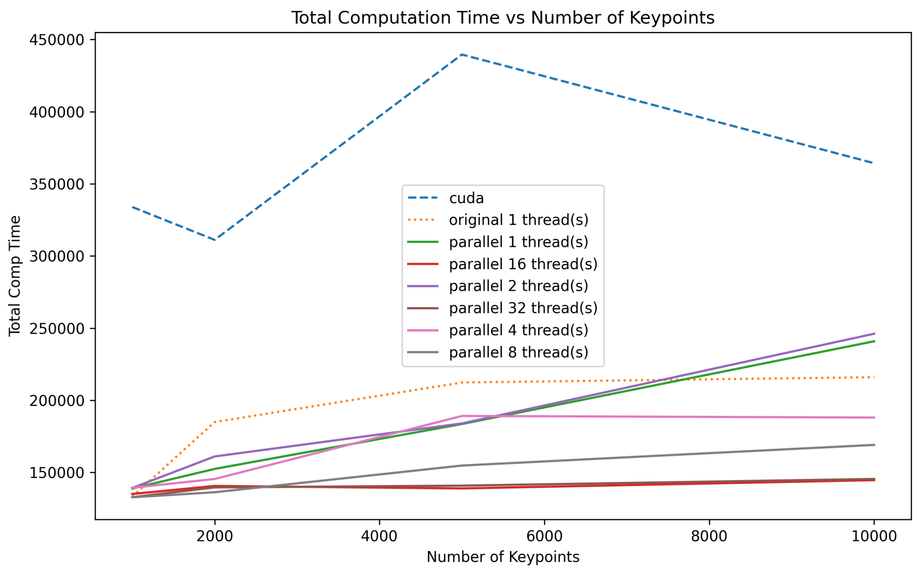
In this article, we focuses on parallelizing the Oriented FAST and Rotated BRIEF (ORB) algorithm with OpenMP. We mainly paralleled the descriptor generating part, which extract and compare 256 pairs of pixels of the given image. We also do a series of experiments to test out the scalability and efficiency of our paralleled version of ORB algorithm.

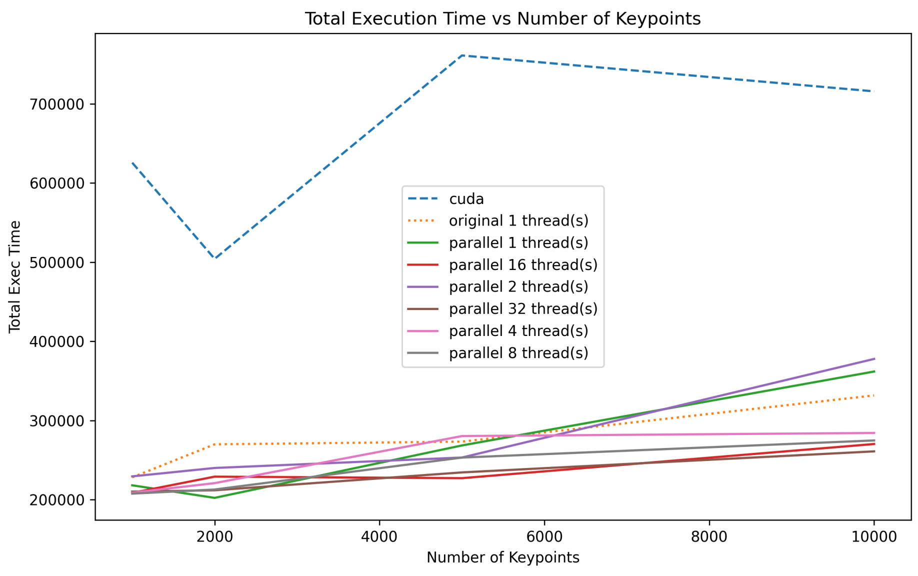
Our experiments reveal that our parallel ORB algorithm has strong scalability and high efficiency, which become increasingly apparent with the growth in the number of keypoints and the number of threads utilized.

Incorporating parallel computing into the BRIEF segment of ORB markedly enhances its performance, particularly for large-scale images. This advancement positions our parallel ORB as a more viable candidate for real-time applications and extensive image processing tasks, maintaining its accuracy and reliability.

[1] Rublee, E., et al, *ORB: An efficient alternative to SIFT or SURF.* (ICCV, 2011).
[2] Tareen, S. A. K., Saleem, Z, *A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK.* (IJACSA, 2018).
[3] Rosten, E., Drummond, T, *Machine learning for high-speed corner detection.* (ECCV, 2006).
[4] Calonder, M., et al, *BRIEF: Binary robust independent elementary features.* (ECCV, 2010).
[5] OpenMP Architecture Review Board, *OpenMP application program interface version 3.0.* (2008)
[6] Filippo Muzzini, Nicola Capodieci, Roberto Cavicchioli, Benjamin Rouxel, *Brief Announcement: Optimized GPU-accelerated Feature Extraction for ORB-SLAM Systems.* (SPAA 2023: 299-302)
[7] cv::cuda::ORB Class Reference


Descriptor Computation Time vs Number of Keypoints
Descr Comp Time
Number of Keypoints
0
100000
200000
300000
400000
2000
4000
6000
8000
10000
cuda
original 1 thread(s)
parallel 1 thread(s)
parallel 16 thread(s)
parallel 2 thread(s)
parallel 32 thread(s)
parallel 4 thread(s)
parallel 8 thread(s)


Total Computation Time vs Number of Keypoints
Total Comp Time
Number of Keypoints
150000
200000
250000
300000
350000
400000
450000
2000
4000
6000
8000
10000
cuda
original 1 thread(s)
parallel 1 thread(s)
parallel 16 thread(s)
parallel 2 thread(s)
parallel 32 thread(s)
parallel 4 thread(s)
parallel 8 thread(s)


Total Execution Time vs Number of Keypoints
Total Exec Time
Number of Keypoints
200000
300000
400000
500000
600000
700000
2000
4000
6000
8000
10000
cuda
original 1 thread(s)
parallel 1 thread(s)
parallel 16 thread(s)
parallel 2 thread(s)
parallel 32 thread(s)
parallel 4 thread(s)
parallel 8 thread(s)