Report

MP.1 Data Buffer Optimization

Implement a vector for `dataBuffer` objects whose size does not exceed a limit (e.g. 2 elements). This can be achieved by pushing in new elements on one end and removing elements on the other end:

This is done by checking the `dataBuffer` size at every iteration, if it's full; then we shift the elements one to the left so that the first element goes to the end. Finally, we replace the last element with the new frame. This way, we implement a vector that retains the last `dataBufferSize` elements in it.

```
// list of data frames which are held in memory at the same time
vector<DataFrame> dataBuffer;

// push image into data frame buffer
DataFrame frame;
frame.cameraImg = imgGray;
if (dataBuffer.size() < dataBufferSize)
{
    dataBuffer.push_back(frame);
}
else
{
    // this shifts elements in the vector one to the left (first element goes to the end)
    rotate(dataBuffer.begin(), dataBuffer.begin() + 1, dataBuffer.end());
    dataBuffer.back() = frame;      // replacing the last element with new frame
}
```

MP.2 Keypoint Detection

Implement detectors `HARRIS`, `FAST`, `BRISK`, `ORB`, `AKAZE`, and `SIFT` and make them selectable by setting a string accordingly:

```
// MidTermProject_Camera_Student.cpp

vector<cv::KeyPoint> keypoints; // create empty feature list for current image
string detectorType = "FAST";   // select of one: [SHITOMASI, HARRIS, FAST, BRISK, ORB, AKAZE, SIFT]

bool bVisKeypoints = false;
if (detectorType.compare("SHITOMASI") == 0)
{
    detKeypointsShiTomasi(keypoints, imgGray, bVisKeypoints);
}
else if (detectorType.compare("HARRIS") == 0)
{
    detKeypointsHarris(keypoints, imgGray, bVisKeypoints);
}
else
{
    detKeypointsModern(keypoints, imgGray, detectorType, bVisKeypoints);
}
```

Harris Keypoint Detector:

```
// matching2D_Student.cpp

void detKeypointsHarris(vector<cv::KeyPoint> &keypoints, cv::Mat &img, bool bVis)
{
  int blockSize = 4;           //  size of an average block for computing a derivative covariation matrix over each pixel neighborhood
  int apertureSize = 5;        //  aperture parameter for the Sobel operator (usually odd number larger than blockSize)
  int k = 0.04;                //  controls the sensitivity of the corner detector (in corner respose R; suggested: 0.04 - 0.06); smaller -> more sensitive -> more corners detected --> more false positives
  int minResponse = 15;        //  minimum value for a corner in the 8bit scaled response matrix
  double maxOverlap = 0.0;     //  max. permissible overlap between two features in %, used during non-maxima suppression
  double t = (double)cv::getTickCount();

  cv::Mat cornerness, cornernessNorm;
  cornerness = cv::Mat::zeros(img.size(), CV_32FC1);
  cv::cornerHarris(img, cornerness, blockSize, apertureSize, k);
  cv::normalize(cornerness, cornernessNorm, 0, 255, cv::NORM_MINMAX, CV_32FC1, cv::Mat());

  // add corners to result vector
  for (int y = 0; y < cornernessNorm.rows; y++)
  {
      for (int x = 0; x < cornernessNorm.cols; x++)
      {
          int response = (int)cornernessNorm.at<float>(y, x);
          if ( response > minResponse)  // only store points above threshold
          {
              cv::KeyPoint newKeyPoint;
              newKeyPoint.pt = cv::Point2f(x, y);
              newKeyPoint.size = 2 * apertureSize;
              newKeyPoint.response = response;

              bool bOverlap = false;
              for (auto item = keypoints.begin(); item != keypoints.end(); ++item)
              {
                  double kptOverlap = cv::KeyPoint::overlap(newKeyPoint, *item);
                  if (kptOverlap > maxOverlap)
                  {
                      bOverlap = true;
                      if (newKeyPoint.response > (*item).response)
                      {
                          // if overlapping and new response is stronger --> use the new keypoint
                          *item = newKeyPoint;
                          break;
                      }
                  }
              }
              if (!bOverlap)
              {
                  // only add keypoints if it's not overlapping
                  keypoints.push_back(newKeyPoint);
              }
          }

      }
  }
  t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
  // cout << "Harris-Corner detection with n=" << keypoints.size() << " keypoints in " << 1000 * t / 1.0 << " ms" << endl;

  // visualize results
  if (bVis)
  {
      cv::Mat visImage = img.clone();
      cv::drawKeypoints(img, keypoints, visImage, cv::Scalar::all(-1), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
      string windowName = "Harris-Corner Detector Results";
      cv::namedWindow(windowName, 7);
      imshow(windowName, visImage);
```

```
        cv::waitKey(0);
    }
}
```

Modern Keypoint Detectors (FAST, BRISK, ORB, AKAZE, SIFT**):**

```
// matching2D_Student.cpp

void detKeypointsModern(std::vector<cv::KeyPoint> &keypoints, cv::Mat &img, std::string detectorType, bool bVis)
{

  cv::Ptr<cv::FeatureDetector> detector;

  if (detectorType.compare("FAST") == 0)
  {
      int threshold = 30;    // difference between intensity of the central pixel and pixels of a circle around this pixel
      bool bNMS = true;      // perform non-maxima suppression on keypoints
      cv::FastFeatureDetector::DetectorType type = cv::FastFeatureDetector::TYPE_9_16; // TYPE_9_16, TYPE_7_12, TYPE_5_8
      // This uses the 16 surrounding pixels to detect whether a pixel is a corner, requiring a contiguous set of 9 out of 16 pixels to be either darker or lighter by the threshold.
      detector = cv::FastFeatureDetector::create(threshold, bNMS, type);
  }
  else if (detectorType.compare("BRISK") == 0)
  {
      detector = cv::BRISK::create();
  }
  else if (detectorType.compare("ORB") == 0)
  {
      detector = cv::ORB::create();
  }
  else if (detectorType.compare("AKAZE") == 0)
  {
      detector = cv::AKAZE::create();
  }
  else if (detectorType.compare("SIFT") == 0)
  {
      detector = cv::SIFT::create();
  }
  else
  {
      throw invalid_argument("Invalid detectorType: " + detectorType + "; should be on of: [FAST, BRISK, ORB, AKAZE, SIFT]");
  }

  double t = (double)cv::getTickCount();
  detector->detect(img, keypoints);
  t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
  // cout << detectorType << " with n= " << keypoints.size() << " keypoints in " << 1000 * t / 1.0 << " ms" << endl;

  // visualize results
  if (bVis)
  {
      cv::Mat visImage = img.clone();
      cv::drawKeypoints(img, keypoints, visImage, cv::Scalar::all(-1), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

      // draw red rectangle
      cv::Rect rect(535, 180, 180, 150); // x, y, width, height
      cv::rectangle(visImage, rect, cv::Scalar(0, 255, 0), 2);

      string windowName = detectorType + " Keypoint Detector Results";
      cv::namedWindow(windowName, 7);
      imshow(windowName, visImage);
      cv::waitKey(0);
  }
}
```

MP.3 Keypoint Removal

Remove all keypoints outside of a pre-defined rectangle and only use the keypoints within the rectangle for further processing:

This is done by looping over the points inside the keypoint vector and checking whether the given rectangle contains the point, if yes add it to the new list and at the end replace the old keypoints vector with the new filtered one.

```
// MidTermProject_Camera_Student.cpp

// only keep keypoints on the preceding vehicle
bool bFocusOnVehicle = true;
cv::Rect vehicleRect(535, 180, 180, 150);
if (bFocusOnVehicle)
{
    // remove keypoints that are outside the rectangle
    vector<cv::KeyPoint> keypointsInsideRect;
    for (auto &kpt : keypoints)
    {
        if (vehicleRect.contains(kpt.pt))
        {
            keypointsInsideRect.push_back(kpt);
        }
    }
    keypoints = keypointsInsideRect;   // replace old keypoints
}
```

MP.4 Keypoint Descriptors

Implement descriptors `BRIEF`, `ORB`, `FREAK`, `AKAZE` and `SIFT` and make them selectable by setting a string accordingly:

```
// MidTermProject_Camera_Student.cpp

cv::Mat descriptors;
string descriptorType = "BRIEF"; // select on of: [BRISK, BRIEF, ORB, FREAK, AKAZE, SIFT]
descKeypoints((dataBuffer.end() - 1)->keypoints, (dataBuffer.end() - 1)->cameraImg, descriptors, descriptorType);
```

```
// matching2D_Student.cpp

void descKeypoints(vector<cv::KeyPoint> &keypoints, cv::Mat &img, cv::Mat &descriptors, string descriptorType)
{
    // select appropriate descriptor: BRISK, BRIEF, ORB, FREAK, AKAZE, SIFT
    cv::Ptr<cv::DescriptorExtractor> extractor;
    if (descriptorType.compare("BRISK") == 0)
    {

        int threshold = 30;        // FAST/AGAST detection threshold score.
        int octaves = 3;           // detection octaves (use 0 to do single scale)
        float patternScale = 1.0f; // apply this scale to the pattern used for sampling the neighbourhood of a keypoint.

        extractor = cv::BRISK::create(threshold, octaves, patternScale);
```

```
    }
    else if (descriptorType.compare("BRIEF") == 0)
    {
        extractor = cv::xfeatures2d::BriefDescriptorExtractor::create();
    }
    else if (descriptorType.compare("ORB") == 0)
    {
        extractor = cv::ORB::create();
    }
    else if (descriptorType.compare("FREAK") == 0)
    {
        extractor = cv::xfeatures2d::FREAK::create();
    }
    else if (descriptorType.compare("AKAZE") == 0)
    {
        extractor = cv::AKAZE::create();
    }
    else if (descriptorType.compare("SIFT") == 0)
    {
        extractor = cv::SIFT::create();
    }
    else
    {
        throw invalid_argument("Invalid descriptorType: " + descriptorType + "; should be on of: [BRISK, BRIEF, ORB, FREAK, AKAZE, SIFT]");
    }

    // perform feature description

    extractor->compute(img, keypoints, descriptors);
}
```

MP.5 Descriptor Matching

Implement **FLANN** matching as well as **K-Nearest-Neighbor** selection. Both methods must be selectable using the respective strings in the main function:

```
// MidTermProject_Camera_Student.cpp

vector<cv::DMatch> matches;
string matcherType = "MAT_FLANN";        // select on of: [MAT_BF, MAT_FLANN]
string selectorType = "SEL_KNN";         // select on of: [SEL_NN, SEL_KNN]

// DES_BINARY, DES_HOG  (this is important when using Brute-Force matching)
if (descriptorType.compare("SIFT") == 0 || descriptorType.compare("AKAZE") == 0)
{
    // SIFT and AKAZE output descriptors as real values
    //  --> hence Norm_L2 should be used to calculate distance between descriptors for matching.
    string descriptorType = "DES_HOG";
}
else
{
    // output binary descriptors (string of 0s and 1s) --> hence Norm_Hamming should be used.
    string descriptorType = "DES_BINARY";
}

matchDescriptors((dataBuffer.end() - 2)->keypoints, (dataBuffer.end() - 1)->keypoints,
                             (dataBuffer.end() - 2)->descriptors, (dataBuffer.end() - 1)->descriptors,
                             matches, descriptorType, matcherType, selectorType);
```

Note: Before applying **FLANN** matcher, both source and reference descriptor should converted to `CV_32F`, this is a workaround a bug in OpenCV.

```
// matching2D_Student.cpp

// Find best matches for keypoints in two camera images based on several matching methods
void matchDescriptors(std::vector<cv::KeyPoint> &kPtsSource, std::vector<cv::KeyPoint> &kPtsRef, cv::Mat &descSource, cv::Mat &descRef,
                      std::vector<cv::DMatch> &matches, std::string descriptorType, std::string matcherType, std::string selectorType)
{
    // configure matcher
    bool crossCheck = false;
    cv::Ptr<cv::DescriptorMatcher> matcher;

    if (matcherType.compare("MAT_BF") == 0)
    {
        int normType = descriptorType.compare("DES_BINARY") == 0 ? cv::NORM_HAMMING : cv::NORM_L2;
        matcher = cv::BFMatcher::create(normType, crossCheck);
        // cout << "BF matching cross-check=" << crossCheck;
    }
    else if (matcherType.compare("MAT_FLANN") == 0)
    {
        // OpenCV bug workaround : convert binary descriptors to floating point due to a bug in current OpenCV implementation
        if (descSource.type() != CV_32F)
        {
            descSource.convertTo(descSource, CV_32F);
        }

        if (descRef.type() != CV_32F)
        {
            descRef.convertTo(descRef, CV_32F);
        }

        // Implement FLANN matching (used L2_Norm by default to create a kd-tree)
        matcher = cv::FlannBasedMatcher::create();
        // cout << "FLANN matching" << endl;
    }

    // perform matching task
    if (selectorType.compare("SEL_NN") == 0)
    { // nearest neighbor (best match)

        matcher->match(descSource, descRef, matches); // Finds the best match for each descriptor in desc1
        // cout << "Nearest-Neighbor (Best Match)" << endl;
    }
    else if (selectorType.compare("SEL_KNN") == 0)
    { // k nearest neighbors (k=2)

        int k = 2;
        vector<vector<cv::DMatch>> knn_matches;
        matcher->knnMatch(descSource, descRef, knn_matches, k);
        // cout << "K-Nearest-Neighbor (Best Match); k=" << k << endl;

        // filter matches using descriptor distance ratio test
        double minDescDistRatio = 0.8;
        for (auto item = knn_matches.begin(); item != knn_matches.end(); ++item)
        {
            if ((*item)[0].distance < minDescDistRatio * (*item)[1].distance)
            {
                // this means that best match has much lower distance than second-best match
                // and most probably is a good match (not a False Positive)
                matches.push_back((*item)[0]);
            }
        }
```

```
    }
}
```

MP.6 Descriptor Distance Ratio

Use the **K-Nearest-Neighbor** matching to implement the descriptor distance ratio test, which looks at the ratio of best vs. second-best match to decide whether to keep an associated pair of keypoints:

The idea is that the good match should have much lower distance than the second-best match; and we found minimum descriptor distance ratio of 0.8 is a good value to differentiate the best from second-best.

We loop over all items in `knn_matches` vector; if the first match is much better than the second match, then most probably it's a true match and hence add it to the final `matches` vector:

```
// matching2D_Student.cpp

// Find best matches for keypoints in two camera images based on several matching methods
void matchDescriptors(std::vector<cv::KeyPoint> &kPtsSource, std::vector<cv::KeyPoint> &kPtsRef, cv::Mat &descSource, cv::Mat &descRef,
                      std::vector<cv::DMatch> &matches, std::string descriptorType, std::string matcherType, std::string selectorType)
{
    ...

    // perform matching task
    if (selectorType.compare("SEL_NN") == 0)
    { // nearest neighbor (best match)

        matcher->match(descSource, descRef, matches); // Finds the best match for each descriptor in desc1
        // cout << "Nearest-Neighbor (Best Match)" << endl;
    }
    else if (selectorType.compare("SEL_KNN") == 0)
    { // k nearest neighbors (k=2)

        int k = 2;
        vector<vector<cv::DMatch>> knn_matches;
        matcher->knnMatch(descSource, descRef, knn_matches, k);
        // cout << "K-Nearest-Neighbor (Best Match); k=" << k << endl;

        // filter matches using descriptor distance ratio test
        double minDescDistRatio = 0.8;
        for (auto item = knn_matches.begin(); item != knn_matches.end(); ++item)
        {
            if ((*item)[0].distance < minDescDistRatio * (*item)[1].distance)
            {
                // this means that best match has much lower distance than second-best match
                // and most probably is a good match (not a False Positive)
                matches.push_back((*item)[0]);
            }
        }
    }
}
```

Performance Evaluation

💡 To satisfy the **requirements in Performance section of Rubric**, another script named `benchmark.cpp` is created which is very similar to the `MidTermProject_Camera_Student.cpp` script but with additional outer for-loop over the list of keypoint detectors and one inner for-loop over the list of keypoint descriptors. The timing over all 10 images are recorded and added to a `.csv` file for later analysis.

`Benchmark` function inside `benchmark.cpp` runs the exact code inside `MidTermProject_Camera_Student.cpp` for a given keypoint detector and descriptor combination and then the results (i.e.: number of detected keypoints, number of matched keypoints, mean size of keypoints, stddev of keypoints, the time takes to detect the keypoints and finally the time takes to detect and describe the keypoints) are saved to data structure `BenchData` and returned from the function and then saved to a `.csv` file.

Note: We've noticed that **`AKAZE`** keypoint detector is only compatible with it's own **`AKAZE`** descriptor. And **`SIFT`** keypoint detector when combined with **`ORB`** keypoint descriptor gives out of memory error.

```
// benchmark2D.hpp

struct BenchData {
    int numDetectKpts;
    int numMatchKpts;
    double timeDetectKpts;
    double timeDetectAndMatchKpts;
    float sizeMeanKpts;
    float sizeStdKpts;
};

std::tuple<float, float> calculate_keypoint_size_statistics(std::vector<cv::KeyPoint> &keypoints);
BenchData benchmark(std::string detectorType, std::string descriptorType, std::string matcherType = "MAT_BF", std::string selectorType = "SEL_KNN", bool bFocusOnVehicle = true, bool bLimitKpts = false);
```

```
// benchmark2D.cpp

tuple<float, float> calculate_keypoint_size_statistics(vector<cv::KeyPoint> &keypoints)
{
    Eigen::VectorXf data(keypoints.size());
    for (int i = 0; i < keypoints.size(); ++i)
    {
        data(i) = keypoints[i].size;        // type of the kpt.size is float
    }
    float mean = data.mean();
    float stddev = sqrt((data.array() - mean).square().sum() / (data.size() - 1));

    return make_tuple(mean, stddev);
}


BenchData benchmark(std::string detectorType, std::string descriptorType, std::string matcherType, std::string selectorType, bool bFocusOnVehicle, bool bLimitKpts)
{

    // same code in MidTermProject_Camera_Student.cpp
    ...

    BenchData benchData;
    benchData.numDetectKpts = accumulate(numDetectKpts.begin(), numDetectKpts.end(), 0);
    benchData.numMatchKpts = accumulate(numMatchKpts.begin(), numMatchKpts.end(), 0);
    benchData.sizeMeanKpts = accumulate(sizeMeanKpt.begin(), sizeMeanKpt.end(), 0.0f) / sizeMeanKpt.size();
    benchData.sizeStdKpts = accumulate(sizeStdKpt.begin(), sizeStdKpt.end(), 0.0f) / sizeStdKpt.size();
    benchData.timeDetectKpts = accumulate(timeDetectKpts.begin(), timeDetectKpts.end(), 0.0);
    benchData.timeDetectAndMatchKpts = accumulate(timeDetectAndDescribeKpts.begin(), timeDetectAndDescribeKpts.end(), 0.0);

    return benchData;
```

```
}


int main()
{
    vector<string> detectorList = {"SHITOMASI", "HARRIS", "SIFT", "FAST", "BRISK", "ORB", "AKAZE"};
    vector<string> descriptorList = {"BRISK", "BRIEF", "ORB", "FREAK", "SIFT", "AKAZE"};

    // Open the CSV file
    std::ofstream file("../benchmark_data.csv");
    file << "detectorType,descriptorType,numDetectKpts,numMatchKpts,timeDetectKpts,timeDetectAndMatchKpts,sizeMeanKpts,sizeStdKpts\n";


    for (string detectorType : detectorList)
    {
        for (string descriptorType : descriptorList)
        {
            try
            {
            cout << "Benchmarking: " << detectorType << " (detector), " << descriptorType << " (descriptor)" << endl;
            BenchData benchmarkData = benchmark(detectorType, descriptorType);

            // Write data immediately after benchmarking
            file << detectorType << "," << descriptorType << ",";
            file << benchmarkData.numDetectKpts << "," << benchmarkData.numMatchKpts << ",";
            file << benchmarkData.timeDetectKpts << "," << benchmarkData.timeDetectAndMatchKpts << ",";
            file << benchmarkData.sizeMeanKpts << "," << benchmarkData.sizeStdKpts << "\n";
            }
            catch (cv::Exception& e)
            {
                std::cerr << "Caught OpenCV exception: " << e.what() << std::endl;
                file << detectorType << "," << descriptorType << ",";
                file << "-,-,-,-,-,-" << "\n";
            }
        }
    }

    return 0;
}
```

MP.7 Performance Evaluation 1

Count the number of keypoints on the preceding vehicle for all 10 images and take note of the distribution of their neighborhood size. Do this for all the detectors you have implemented.

Check the code above to see how this implemented. Here are the results in table below and saved to `benchmark_data.ods`:

detectorType	descriptorType	numDetectKpts	sizeMeanKpts	sizeStdKpts
SHITOMASI	BRISK	1179	4	0
SHITOMASI	BRIEF	1179	4	0
SHITOMASI	ORB	1179	4	0
SHITOMASI	FREAK	1179	4	0
SHITOMASI	SIFT	1179	4	0
SHITOMASI	AKAZE	-	-	-
HARRIS	BRISK	262	10	0
HARRIS	BRIEF	262	10	0
HARRIS	ORB	262	10	0
HARRIS	FREAK	262	10	0
HARRIS	SIFT	262	10	0
HARRIS	AKAZE	-	-	-
SIFT	BRISK	1386	5.03235	5.96749
SIFT	BRIEF	1386	5.03235	5.96749
SIFT	ORB	-	-	-
SIFT	FREAK	1386	5.03235	5.96749
SIFT	SIFT	1386	5.03235	5.96749
SIFT	AKAZE	-	-	-
FAST	BRISK	1491	7	0
FAST	BRIEF	1491	7	0
FAST	ORB	1491	7	0
FAST	FREAK	1491	7	0
FAST	SIFT	1491	7	0
FAST	AKAZE	-	-	-
BRISK	BRISK	2762	21.9422	14.6079
BRISK	BRIEF	2762	21.9422	14.6079
BRISK	ORB	2762	21.9422	14.6079
BRISK	FREAK	2762	21.9422	14.6079
BRISK	SIFT	2762	21.9422	14.6079
BRISK	AKAZE	-	-	-
ORB	BRISK	1161	56.0578	25.246
ORB	BRIEF	1161	56.0578	25.246
ORB	ORB	1161	56.0578	25.246
ORB	FREAK	1161	56.0578	25.246
ORB	SIFT	1161	56.0578	25.246
ORB	AKAZE	-	-	-
AKAZE	BRISK	1670	7.69342	3.54178
AKAZE	BRIEF	1670	7.69342	3.54178
AKAZE	ORB	1670	7.69342	3.54178
AKAZE	FREAK	1670	7.69342	3.54178
AKAZE	SIFT	1670	7.69342	3.54178
AKAZE	AKAZE	1670	7.69342	3.54178

MP.8 Performance Evaluation 2

Count the number of matched keypoints for all 10 images using all possible combinations of detectors and descriptors. In the matching step, the BF approach is used with the descriptor distance ratio set to 0.8:

detectorType	descriptorType	numMatchKpts
SHITOMASI	BRISK	690
SHITOMASI	BRIEF	816
SHITOMASI	ORB	765
SHITOMASI	FREAK	575
SHITOMASI	SIFT	927
SHITOMASI	AKAZE	-
HARRIS	BRISK	177
HARRIS	BRIEF	193
HARRIS	ORB	180
HARRIS	FREAK	169
HARRIS	SIFT	194
HARRIS	AKAZE	-
SIFT	BRISK	536
SIFT	BRIEF	597
SIFT	ORB	-
SIFT	FREAK	500
SIFT	SIFT	800
SIFT	AKAZE	-
FAST	BRISK	776
FAST	BRIEF	883
FAST	ORB	859
FAST	FREAK	657
FAST	SIFT	1046
FAST	AKAZE	-
BRISK	BRISK	1298
BRISK	BRIEF	1344
BRISK	ORB	913
BRISK	FREAK	1090
BRISK	SIFT	1646
BRISK	AKAZE	-
ORB	BRISK	649
ORB	BRIEF	450
ORB	ORB	515
ORB	FREAK	349
ORB	SIFT	763
ORB	AKAZE	-
AKAZE	BRISK	1110
AKAZE	BRIEF	1087
AKAZE	ORB	922
AKAZE	FREAK	966
AKAZE	SIFT	1270
AKAZE	AKAZE	1172

MP.9 Performance Evaluation 3

Log the time it takes for keypoint detection and descriptor extraction. The results must be entered into a spreadsheet and based on this data, the TOP3 detector / descriptor combinations must be recommended as the best choice for our purpose of detecting keypoints on vehicles.

detectorType	descriptorType	timeDetectKpts	timeDetectAndMatchKpts
SHITOMASI	BRISK	71.8241	261.436
SHITOMASI	BRIEF	62.5354	68.047
SHITOMASI	ORB	68.6499	75.9609
SHITOMASI	FREAK	49.3851	222.187
SHITOMASI	SIFT	43.0793	114.218
SHITOMASI	AKAZE	-	-
HARRIS	BRISK	88.9418	271.862
HARRIS	BRIEF	83.6372	84.977
HARRIS	ORB	83.3443	88
HARRIS	FREAK	82.5348	242.711
HARRIS	SIFT	82.4422	152.297
HARRIS	AKAZE	-	-
SIFT	BRISK	535.085	637.611
SIFT	BRIEF	533.059	536.824
SIFT	ORB	-	-
SIFT	FREAK	451.818	617.519
SIFT	SIFT	443.587	804.481
SIFT	AKAZE	-	-
FAST	BRISK	5.07279	198.746
FAST	BRIEF	5.6976	9.08532
FAST	ORB	5.63022	12.5465
FAST	FREAK	5.61386	172.694
FAST	SIFT	5.18878	78.3605
FAST	AKAZE	-	-
BRISK	BRISK	408.784	606.8
BRISK	BRIEF	410.783	415.239
BRISK	ORB	412.028	439.414
BRISK	FREAK	410.17	578.43
BRISK	SIFT	410.488	520.701
BRISK	AKAZE	-	-
ORB	BRISK	117.963	314.853
ORB	BRIEF	42.3118	44.9663
ORB	ORB	39.1927	65.2617
ORB	FREAK	38.8232	201.734
ORB	SIFT	40.1016	178.645
ORB	AKAZE	-	-
AKAZE	BRISK	240.969	438.037
AKAZE	BRIEF	241.224	245.672
AKAZE	ORB	238.627	256.649
AKAZE	FREAK	243.84	414.636
AKAZE	SIFT	238.24	334.527
AKAZE	AKAZE	238.446	436.795

All the Results:

detectorType	descriptorType	numDetectKpts	numMatchKpts	timeDetectKpts	timeDetectAndMatchKpts	sizeMeanKpts	sizeStdKpts
SHITOMASI	BRISK	1179	690	71.8241	261.436	4	0
SHITOMASI	BRIEF	1179	816	62.5354	68.047	4	0
SHITOMASI	ORB	1179	765	68.6499	75.9609	4	0
SHITOMASI	FREAK	1179	575	49.3851	222.187	4	0
SHITOMASI	SIFT	1179	927	43.0793	114.218	4	0
SHITOMASI	AKAZE	-	-	-	-	-	-
HARRIS	BRISK	262	177	88.9418	271.862	10	0
HARRIS	BRIEF	262	193	83.6372	84.977	10	0
HARRIS	ORB	262	180	83.3443	88	10	0
HARRIS	FREAK	262	169	82.5348	242.711	10	0
HARRIS	SIFT	262	194	82.4422	152.297	10	0
HARRIS	AKAZE	-	-	-	-	-	-
SIFT	BRISK	1386	536	535.085	637.611	5.03235	5.96749
SIFT	BRIEF	1386	597	533.059	536.824	5.03235	5.96749
SIFT	ORB	-	-	-	-	-	-
SIFT	FREAK	1386	500	451.818	617.519	5.03235	5.96749
SIFT	SIFT	1386	800	443.587	804.481	5.03235	5.96749
SIFT	AKAZE	-	-	-	-	-	-
FAST	BRISK	1491	776	5.07279	198.746	7	0
FAST	BRIEF	1491	883	5.6976	9.08532	7	0
FAST	ORB	1491	859	5.63022	12.5465	7	0
FAST	FREAK	1491	657	5.61386	172.694	7	0
FAST	SIFT	1491	1046	5.18878	78.3605	7	0
FAST	AKAZE	-	-	-	-	-	-
BRISK	BRISK	2762	1298	408.784	606.8	21.9422	14.6079
BRISK	BRIEF	2762	1344	410.783	415.239	21.9422	14.6079
BRISK	ORB	2762	913	412.028	439.414	21.9422	14.6079
BRISK	FREAK	2762	1090	410.17	578.43	21.9422	14.6079
BRISK	SIFT	2762	1646	410.488	520.701	21.9422	14.6079
BRISK	AKAZE	-	-	-	-	-	-
ORB	BRISK	1161	649	117.963	314.853	56.0578	25.246
ORB	BRIEF	1161	450	42.3118	44.9663	56.0578	25.246
ORB	ORB	1161	515	39.1927	65.2617	56.0578	25.246
ORB	FREAK	1161	349	38.8232	201.734	56.0578	25.246
ORB	SIFT	1161	763	40.1016	178.645	56.0578	25.246
ORB	AKAZE	-	-	-	-	-	-
AKAZE	BRISK	1670	1110	240.969	438.037	7.69342	3.54178
AKAZE	BRIEF	1670	1087	241.224	245.672	7.69342	3.54178
AKAZE	ORB	1670	922	238.627	256.649	7.69342	3.54178
AKAZE	FREAK	1670	966	243.84	414.636	7.69342	3.54178
AKAZE	SIFT	1670	1270	238.24	334.527	7.69342	3.54178
AKAZE	AKAZE	1670	1172	238.446	436.795	7.69342	3.54178

💡 **Based on this Results; TOP3 recommendations:**

1. BRISK keypoint detector has the most detected keypoints and when combined with **SFIT** has the most number of matched keypoints while having also decent size distribution of the detected keypoints.

2. BRISK-BRIEF is the second best combination that gives the most detected and matched keypoints.

3. Real-Time Application: for real time applications, we can consider the combination of **FAST-BRIEF & FAST-ORB** where more than 800 keypoints were matched in around 10 (ms).