CUDA-Accelerated Feature-Based Egomotion Estimation

Safa Ouerghi, Rémi Boutteau, Xavier Savatier, Fethi Tlili

CUDA-accelerated Feature-based Egomotion Estimation

Safa Ouerghi[1], Remi Boutteau[2], Xavier Savatier[2], and Fethi Tlili[1]

[1] Carthage Univ, SUP'COM, GRESCOM, 2083 El Ghazela, Tunisia
{safa.ouerghi,fethi.tlili}@supcom.tn
[2] Normandie Univ, UNIROUEN, ESIGELEC, IRSEEM, 76000 Rouen, France
{Remi.Boutteau,Xavier.Savatier}@esigelec.fr

Abstract. Egomotion estimation is a fundamental issue in structure from motion and autonomous navigation for mobile robots. Several camera motion estimation methods from a set of variable number of image correspondences have been proposed. Seven- and eight- point methods have been first designed to estimate the fundamental matrix. Five-point methods represent the minimal number of required correspondences to estimate the essential matrix. These feature-based methods raised special interest for their application in a hypothesize-and-test framework to deal with the problem of outliers. This algorithm allows relative pose recovery at the expense of a much higher computational time when dealing with higher ratios of outliers. To solve this problem with a certain amount of speedup, we propose in this work, a CUDA-based solution for the essential matrix estimation from eight, seven and five point correspondences, complemented with robust estimation. The mapping of these algorithms to the CUDA hardware architecture is given in detail as well as the hardware-specific performance considerations. The correspondences in the presented schemes are formulated as bearing vectors to be able to deal with all camera systems. Performance analysis against existing CPU implementations is also given, showing a speedup 4 times faster than the CPU for an outlier ratio $\epsilon = 0.5$ which is common for the essential matrix estimation from automatically computed point correspondences, for the five-point-based estimation. More speedup was shown for the seven and eight- point based implementations reaching 76 times and 57 times respectively.

Keywords: Egomotion · visual odometry · robotics · CUDA · GPU.

1 Introduction

Accurate localization is a fundamental issue in autonomous navigation that has been extensively studied by the Robotics community. First advancements have emerged within the Simultaneous Localization And Mapping (SLAM) approach -the process of generating an internal map using sensor observations while moving through an environment- has received a great deal of attention [1]. Many sensory devices have been used in SLAM systems including lidar scanners and

cameras that have become very popular due to their low-cost, wide availability, passive nature and modest power requirements [2, 3]. Vision-based SLAM systems coined Visual SLAM (or v-SLAM) [4–6] may include several modules to concurrently perform the tracking and mapping tasks such as a visual odometry module (VO) [7, 8]. This latter is derived from Structure from Motion (Sfm) and refers to the process of incrementally estimating the egomotion of an agent (e.g., vehicle, human and robot) using only the input of a single or multiple cameras attached to it. The feature-based VO for the monocular scheme consists mainly in finding corresponding features in consecutive frames in the video sequence and using the scene's epipolar geometry to calculate the position and orientation changes between the two images. A common way of determining the relative pose using two images taken by a calibrated camera is based on the estimation of the essential matrix that has been studied for decades. The first efficient implementation of the essential matrix estimation is proposed by Nister in [9] and uses only five point correspondances. The work of Stewenius built upon the work of Nister uses the Gröbner Basis to enhance the estimation accuracy [10]. Although the essential matrix expresses the epipolar geometry between two views taken by a calibrated camera, a more general relationship can be derived in the case of a non-calibrated camera expressed by the fundamental matrix which is is the algebraic representation of epipolar geometry [11]. However, in a real application, wrong matches can lead to severe errors in the measurements, which are called outliers and that occurs during the descriptors matching step. The typical way of dealing with outliers consists of first finding approximate model parameters by iteratively applying a minimal solution in a hypothesize-and-test scheme. This procedure allows us to identify the inlier subset, and then, a least-squares result is obtained by minimizing the reprojection error of all inliers via a linear solution or a non-linear optimization scheme, depending on the complexity of the problem. This scheme is called RANdom Sample Consensus (RANSAC) and has been first proposed by Fischler and Bolles [12]. RANSAC can often find the correct solution even for high levels of contamination. However, the number of samples required to do so increases exponentially, and the associated computational cost is substantial. Especially for robotics systems, the challenges are more acute, due to their stringent time-response requirements. To solve these problems with a certain amount of speedup, the usage of GPU computation is a popular topic in the community. Researchers and developers have become interested in harnessing the Graphics Processing Units (GPUs) power for general-purpose computing, an effort known collectively as GPGPU (for General-Purpose computing on the GPU). The Compute Unified Device Architecture (CUDA) has enabled graphics processors to be explicitly programmed as general-purpose shared-memory multi-core processors with a high level of parallelism [13]. In fact, recently, many problems are being solved using programmable graphics hardware including feature matching and triangulation [14], feature detectors [15], large non-linear optimization problems such as bundle adjustment [16] and learning algorithms [17]. In this paper, we focus on an efficient implementation of a state-of-the-art relative pose estimation

based on the computation of the Essential matrix from five correspondances. We consider single GPU implementation and we describe the strategies to map the problem to CUDA architecture. Futhermore, new Kepler and Maxwell architecture features are used and analyzed, such as CUDA Dynamic Parallelism and new CuBLAS batched interfaces. The outline of this paper is as follows: we briefly present the theory underlying the essential and fundamental matrices estimation in Section 2. Section 3 details the CUDA based implementation of the essential matrix estimation algorithm within RANSAC from five, seven and eiht points. Afterwards, section 4 shows several experiments as examples of the speedup results obtained with our implementation. Finally section 6 gives the conclusion of the paper.

2 Background

The geometric relations between two images of a camera are described by the fundamental matrix and by the essential matrix in case the camera is calibrated. The essential matrix directly holds the parameters of the motion undergone. In this section, we provide an overview of the important background underlying the robust essential matrix estimation as well as the main feature-based methods used to derive the essential matrix and consequently the camera motion.

2.1 Fundamental and Essential matrices

The epipolar geometry exists between any two camera systems. For a point u_i in the first image, its correspondence in the second image, $u_i^{'}$, must lie on the epipolar line in the second image. This is known as the *epipolar constraint*. Algebraically, for u_i and $u_i^{'}$ to be matched, the following equation must be satisfied

$$u_i^{'T} F u_i = 0, \tag{1}$$

where F is the 3×3 fundamental matrix that has seven degrees of freedom because it is of rank 2 (*i.e.* $det(F) = 0$) and is defined only up to a scale. Therefore, at least seven point correspondences are required to calculate it. In fact, each point match gives rise to one linear equation in the unknown entries of F. From all the point matches, we obtain a set of linear equations of the form

$$A F = 0, \tag{2}$$

where A is a 7×9 constraint matrix. If $rank(A) = 8$ when using 8 correspondences, the solution is unique (up to a scale) and can be found by linear methods. However, least squares methods are preferred because of the presence of image noise and quantization effects.

The essential matrix is a 3×3 matrix as well, and can be considered as a special case of the fundamental matrix, satisfying the following relationship

$$x_i^{'T} E x_i = 0, \tag{3}$$

where x_i and $x_i^{'}$ are normalized image point correspondences (*i.e.* $x_i = \mathbf{K}^{-1}u_i$ and $x'_i = \mathbf{K}^{-1}u'_i$ with $\mathbf{K}$ the intrinsic calibration matrix).

The fundamental and the essential matrices are, therefore, related by

$$E = \mathbf{K}^{-1}\, F\, \mathbf{K}, \tag{4}$$

where $\mathbf{K}$ is the intrinsic calibration matrix.

Furthermore, if the two views have relative pose $[R|t]$ then

$$E = [t]_{\times} R, \tag{5}$$

where $[t]_{\times}$is the skew-symmetric matrix with the property that $[t]_{\times} x = t \times x$.

However, from two images alone, the length of t cannot be determined as E is only determined up to a scale.

To direcly compute the essential matrix, expanding Equation 3 is, generally, done which gives a single linear constraint in the nine elements of E for every correspondence. From N correspondences, these equations can be stacked to form a $9 \times N$ matrix which null space obtained by singular value decomposition (SVD) gives a basis for the space in which E lies. The points within this vector space which are essential matrices are those which can be decomposed into a rotation and a translation. E can be decomposed in this way using an SVD decomposition

$$E = U \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 0 \end{pmatrix} V^T, \tag{6}$$

which is equivalent to the following constraint providing an efficient test whether a matrix is approximately an essential matrix

$$EE^T E - \frac{1}{2} trace(EE^T)E = 0. \tag{7}$$

2.2 Essential matrix Computation from feature correspondences

Computing E from eight correspondences F can only be determined up to a scale, there are, thus, 8 unknowns and at least 8 point matchings are used by the *8-point algorithm*. The constraint matrix A obtained according to Equation 2 is, therefore, 9×8. The least square solution is the singular vector corresponding the smallest singular value of A, i.e. the last column of V in the SVD where $A = U\, D\, V^T$.

Computing E from seven correspondences The matrix of constraints A is constructed from $n = 7$ correspondences and will therefore have a two-dimensional null-space. This latter is derived from the use an SVD decomposition resulting in two independent vectors f_1 and f_2. Taking f as the vector containing the coefficients of the fundamental matrix F,

$$f = (F_{11}\, F_{12}\, F_{13}\, F_{21}\, F_{22}\, F_{23}\, F_{31}\, F_{32}\, F_{33})^T, \tag{8}$$

f can be written as $f = (1-t)\, f_1 + t\, f_2$ with $t \in \mathbb{R}$, solution of $A\, f = 0$.

The corresponding F-matrices are therefore,

$$F = (1-t)\, F_1 + t\, F_2. \tag{9}$$

The condition $det(F) = 0$ leads to the equation,

$$det((1-t)\, F_1 + t\, F_2) = 0, \tag{10}$$

that is a 3^{rd} degree polynomial in t having at most 3 real roots, *i.e.* one or three solutions are possible. The correct one is obtained after disambiguation to check for the correct fundamental matrix. In our case, we are dealing with normalized data and would , therefore, directly obtain the essential matrix $E = F$. In fact, the essential and fundamental matrices are related as follows:

$$E = K^{'T}\, F\, K. \tag{11}$$

Computing E from five correspondances Several algorithms have been developed to estimate the essential matrix, including, the seven- and eight-point algorithms that are relatively fast [11]. However, for their use within RANSAC, essential matrix computations have relied on minimal subsets, which for essential matrix is five correspondences. Furthermore, Essential matrix estimation from five correspondances have shown a better accuracy than other faster algorithms with more correspondances. In essential matrix estimation, given five correspondences, four basis vectors satisfying Equation 1 can be computed by SVD. All linear combinations of these basis vectors satisfying Equation 3 are essential matrices that provide nine cubic constraints in the elements of E. The methods of Nister [9], and Stewenius et al. [10] both work by solving these nine equations. Stewenius et al. first showed that the equations can be written as

$$M\, X = 0, \tag{12}$$

where M is a 10×20 matrix.

After gauss-jordan elimination, the system can be written

$$[I\, B]\, X = 0, \tag{13}$$

where I is a 10×10 identity matrix and B a 10×10 matrix.

Stewenius et al. used, subsequently, the action matrix concept to solve the systems in which a Gröbner basis is found. The 10×10 action matrix real eigenvalues and eigenvectors contain, hence, the solutions of polynomial equations.

2.3 Relative pose computation from Essential matrices solutions

Once the essential matrices solutions are computed, they have to be decomposed into rotation and translation. In fact, the decomposition follows the normal procedure for the general case [9], giving two possible solutions for the rotation, R_a

and R_b, and two solutions for the translation as well, t_a and t_b, which have the same direction $\hat{t}$ determined up to a scale.

Thus, if $E \sim USV^T$is the SVD of E, a matrix D is defined as

$$D = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{14}$$

Then, $R_a = UDV^T$ and $R_b = UD^TV^T$. The solution for the translation direction is $\hat{t} = [U_{13}U_{23}U_{33}]^T$.

Four pose configurations are, therefore, obtained for each essential matrix namely, (R_a, t_a), (R_b, t_a), (R_a, t_b) and (R_b, t_b). Consequently, a disambiguation has to be performed to output the correct movement undergone by the camera.

2.4 Robust estimation of the Essential matrix

Even if the underlying dataset is contaminated with outliers, RANSAC estimator can be used to robustly estimate the model parameters. RANSAC uses a randomly chosen subset of the entire dataset to compute a hypothesis. The remaining datapoints are used for validation. Repeating the hypothesis computation and validation with different subsets, the probability of finding a hypothesis that fits the data well increases. For a data set with a given proportion ϵ of outliers, the number of trials N required to give sufficiently high probability p to pick an outlier-free subset consisting of k point correspondences is

$$N = \frac{\log(1-p)}{\log(1-(1-\epsilon)^k)} \tag{15}$$

Since the confidence p is generally chosen to be $p \geq 0.99$, the number of required RANSAC iterations N only depends on the number of parameters k and the assumed ratio of outliers ϵ. Usually, the ratio of outliers ϵ is unknown. Hence, we resort to an adaptive version of RANSAC, where, after each iteration, the number of inliers γ is counted and the outlier ratio is updated according to

$$\epsilon = 1 - \frac{\gamma}{n}, \tag{16}$$

with n equal to the size of the dataset. The number of iterations N is therefore updated based on ϵ.

3 CUDA-based relative motion estimation from 2-D to 2-D correspondences

The complexity of implementing existing algorithms on the GPU depends heavily on control flow of the algorithm. In fact, some algorithms (such as basic image processing) can be classified as *embarrassingly parallel* when little effort is required to split up the process into parallel tasks and are often easily ported

to the GPU. In contrast, other algorithms are *inherently sequential* which implies unexpected scheduling difficulties that prohibit parallelization on the one hand and greatly increases the effort required to implement an efficient CUDA solution on the other hand. As has been explicited in the previous section, relative motion estimation algoritms generally execute within RANSAC to deal with the problem of outliers. Our parallelization approach is based on performing the required RANSAC iterations in parallel on CUDA to achieve a certain amount of speedup. This level of parallelism suggests the consideration of RANSAC iterations as a batch of parallel computations, each processing a small subset of data. However, RANSAC is iherently sequential which puts an additional complexity on the development process. Furthermore, we have relied on the use of CuBLAS, a high-performance implementation of BLAS-3 routines, for linear algebra computations [18]. As the matrices sizes in our problem are below 32×32, we have particularly exploited the batched interface of the CuBLAS library where many small dense matrices factorizations, to be performed simultaneously, are provided. In this section, we present the implementation details of relative pose estimation from five, seven and eight correspondences using the CUDA programming model.

3.1 CUDA-based Relative Motion Estimation from five Correspondences

In this section, we present the implementation details of the essential matrix estimation from five correspondences within RANSAC presented first in [20]. As has been stated before, the eigenvalues of the action matrix contain the essential matrices solutions according to Stewenius's method [10]. However, a device based eigenvalue computation on CUDA doesnt exist yet. Hence, we have relied on the Matlab code provided by Chris Engels, based on the reduction to a single polynomial [21]. This is done through the computation of the action matrix characteristic polynomial roots, equivalent to the action matrix eigenvalues. In total, four kernels have been employed operating at different levels of parallelism. The first, exploits the CuBLAS library batched interface, manages algebraic computations. It employs, therefore, a thread level parallelism and a nested warp level parallelism as it uses dynamic parallelism to call CuBLAS functions from within device. The second employs a straightforward parallelization and works at a thread-level parallelism where each thread manages the remainder computations after the completion of the first kernel, i.e. one thread per RANSAC iteration. The third kernel is used to rate the models outputted by the previous kernel and works at a block level parallelism where each block validates a model relative to one RANSAC iteration. Finally, an additional kernel is used to compute RANSAC's best model and it simply performs a reduction to find the model with maximum number of inliers which represents the best model.

CuBLAS based kernel This kernel is launched with one block and a number of threads equal to the number of required RANSAC iterations. In fact, according

to Equation 15, for a probability of success of 0.99 and a rate of outliers equal to 0.5 the number of RANSAC trials required for a robust estimation based on five points is equal to 145. We assume that the data is contaminated with 0.5% of outliers which is quite common for the essential matrix estimation from automatically computed point correspondences. 145 threads belonging to one block are therefore used. The high level interface exposed by all implementations in this kernel is CuBLAS batched interface for solving a batch of N different systems. Besides the batch size and the matrix dimensions, the functions expect pointers to array of matrices. All arrays are assumed to be stored contiguously with a column major layout and accessed to in global memory through the handle of an array of pointers that we statically allocate as follows:

```
__device__ double* PointersArray[MaxBatchSize]
```

Firstly, a 9×5 hypothesis $A[i]$, $i = 0...batchSize - 1$ is computed from each random five correspondances by each thread. The computed hypotheses are written to global memory and referenced by an array of pointers as indicated above.

Secondly, the null-space of each hypothesis have to be computed by SVD. However, due to the absence of a GPU-based implementation of SVD decomposition, we use instead a QR decomposition to derive the null space. In fact, standard methods for determining the null-space of a matrix are to use a QR decomposition or an SVD. If accuracy is paramount, the SVD is preferred but QR is faster. Using a QR decomposition, if $A^T = QR$, and the rank of A is r, then the last $n - r$ columns of Q make up the null-space for A. This is performed through a call to the cuBLAS built-in function $cublasDqrfBatched$ performing a QR factorization of each $A[i]$ for $i = 0$, ..., $batchSise - 1$. The decomposition output is presented in a packed format where the matrix R is the upper triangular part of each $A[i]$ and the vectors v on the lower part are needed to compute the elementary reflectors. the matrix Q is, hence, not formed explicitly, but is represented as a product of these elementary reflectors.

As CuBLAS dosen't provide a built-in routine to retrieve Q as Lapack does, we designed a child kernel called from the main kernel to simulaneously calculate the different reflectors and compute their product to retrieve Q.

The number of Thread-blocks in the launch configuration of the child kernel is equal to the $batchSize$, *i.e. iterations*. Each Thread-block computes a single matrix Q and a block-level parallelism is hence applied. The Thread-blocks are designed to be three-dimensional, where the *x-dimension* refers to the number of rows of each reflector, the *y-dimension* to the number of columns and the *z-dimension* to the number of reflectors. This allows each thread to handle one element in shared memory and consequently, ensure a parallel computation of the different reflectors. It is worth noting that this configuration is possible because the matrix sizes in our problem are small (5 refrectors, each of size 9×9) and consequently, all reflectors fit at once in shared memory. The computation consists in loading, first, the $A[blockIdx.x]$, and the array of scalars $Tauarray[blockIdx.x]$ exited by $cublasDqrfBatched$ into shared memory where the matrix Q is also allocated. The vector v_i relative to each reflector q_i is then putted in the required form, where $v_i(1 : i - 1) = 0$ and $v_i(i) = 1$ with

$v_i(i+1:m)$ on exit in $A[blockIdx.x][i+1:m,i]$. Each reflector q_i has the form $q_i = I - Tau[i].v.transpose(v)$, computed for all reflectors by the pseudocode explicited in Figure 1 and finally, the product of all reflectors is computed to retrieve Q.

Pseudocode1: Q computation in shared memory

```
int tidx=threadIdx.x;
int tidy=threadIdx.y;
int tidz=threadIdx.z;
int index_A=tidz*9+tidy;
int index_q=tidx*9+tidy+9*9*tidz;
  Q[index_q]=A[index_A];
__syncthreads();
double alpha;alpha=-1;
int index=tidx*9+tidy+9*9*tidz;
  Q[index]= (-Tau[tidz]*Q[index]
               *(Q[tidxx*9+tidy+9*9*tidz]));
__syncthreads();
```

Fig. 1. Pseudocode of reflectors computation in shared memory.

Once the null-space determined, the second step is to compute a 10×20 matrix M that is accelerated in the provided *openSource* code, through a symbolic computation of the expanded constraints. The matrix columns are then rearranged according to a predefined order. To save execution time and memory usage, we use to rearrange the matrix columns beforehand and to write it in column major for subsequent use of cuBLAS functions. We hence output a permuted 20×10 matrix M .

Subsequently, the Reduced Row Echelon Form (RREF) of M have to be computed through a gauss-jordan elimination, *i.e.* $M = [I\ B]$. Instead of carrying out a gauss-jordan elimination on M, a factorization method may be used to find directly the matrix B from the existant matrix M. In fact, cuBLAS provides several batched interfaces for linear systems factorizations. We exploit the batched interface of LU factorization performing four GPU kernel calls for solving systems in the form ($MX = b$) as follows:

1. LU decomposition of M ($P\,M = LU$).
2. Permutation of the array b with the array of pivots P ($y = P\,b$).
3. Solution of the triangular lower system ($L\,c = y$).
4. Solution of the upper system to obtain the final solution ($U\,x = c$)

With putting b as an array of pointers to null vector, cuBLAS directly provides *cublasDgetrfBatched* for the first step and *cublasDgetrsBatched* for the three other steps. We finally obtain the matrix B in exit of *cublasDgetrsBatched,* solution of the system $MX = 0$.

RANSAC models computation kernel At this level, the kernel is launched with one CUDA block and *iterations* number of threads. We only use global memory where the computations of the previous kernel are stored and small per thread arrays using registers and local memory.

Each thread computes a 10^{th} degree polynomial using local variables. This is done by extracting from the RREF in global memory the coefficients of two 3^{rd} degree polynomials and a 4^{th} degree polynomial represented by private local arrays for each thread. These polynomials are afterwards convoluted then substracted and added to generate a single 10^{th} degree polynomial for each thread as explicited in the original Matlab code and which refers to the computation of the determinant of the characteristic polynomial. The convolution is performed in our implementation through a special *device* function presented as a symbolic computation of three polynomials of 3^{rd}, 3^{rd} and 4^{th} degrees respectively.

The key implementation of this kernel is the resolution of a batch of 10^{th} degree polynomials. In fact, we used a batched version of the Durand-Kerner Method in which we assign to each polynomial a thread. We start by giving a brief overview of the Durand-Kerner method, followed by our implementation details.

Durand-Kerner Method The Durand_Kerner Method allows the extraction of all roots ω_1, ..., ω_n of a polynomial

$$p(z) = \sum_{i=0}^{n} a_i z^{n-i}, \tag{17}$$

where $a_n \neq 0$, $a_0 = 1$, $a_i \in \mathbb{C}$.

This method constructs a sequence, $H(z^k) = z^{k+1}$ in $\mathbb{C}^{\mathbb{N}}$ with $Z^{(0)}$being any initial vector and H is the Weierstrass operator making $Z_i^{(k)}$ tends to the root ω_i of the polynomial, defined as:

$$H_i(z) = z_i - \frac{P(z_i)}{\prod_{j \neq i}(z_i - z_j)} \qquad i = 1, ..., n \tag{18}$$

The iterations repeat until $\frac{|Z_i^k - Z_i^{k+1}|}{Z_i^k}$ or $|P(z_i^k)|$ is smaller than the desired accuracy.

GPU version of batched Durand-Kerner The implementation of the Durand-Kerner on GPU, is basically sequential where each thread computes the ten complex roots of the 10^{th} degree polynomial. We defined the type COMPLEX denoting structs of complex numbers. We started from an initial complex guess z randomly chosen, and the vector of complex roots R of size the number of roots (10 in our problem) where, $R[i] = z^i$, $i = 1..n-1$. The function*poly* evaluates at z a polynomial of the form of Equation 17 where the vector $A = a1, a2, a3, ..., a(n-2), a(n-1), a(n)$ denotes the coeffecients of our polynom.

As we are dealing with complex numbers, complex arithmetic has been employed denoted by *compsubtract* for complex numbers substraction and *compdiv* for complex division. As explicited in the following piece of code, we iterate until obtaining the desired accuracy expressed as a relative error of estimated roots below a predefined value as depicted in Figure 2.

Pseudocode2: GPU Version of Durand-Kerner method

```
double maxDiff = 0; int iter=0; int maxIters =30;
for( iter = 0; iter < maxIters; iter++ ) {
  maxDiff = 0;
  for (int j = 0; j < n; j ++) {
    COMPLEX B = poly(A, n, R[j]);
    for (int k = 0; k < n; k++) {
      if (k != j)
          B = compdiv(B,compsubtract(R[j] , R[k]));
       }
      R[j] = compsubtract(R[j],B);
      maxDiff = max(maxDiff, abs(B.x));
   }
  if( maxDiff <= 1e-10)
     break;
  }
```

Fig. 2. Pseudocode of batched Durand-Kerner method on CUDA.

As explicited in Section 2.3, an SVD decomposition of the directly obtained essential matrices which are up to 10 (real solutions of 10^{th} degree polynomial) is used to decompose each solution into rotation and translation. This operation can take a significant portion of the computation time and we use, therefore, a specifically tailored singular value decomposition for essential matrices according to Equation 6, that is proposed in [9] (Appendix B). In our implementation, each thread computes up to 10 essential matrices, and for each, four movement configurations are obtained.

However, in order to deal with all central camera models including perspective, dioptric, omnidirectional and catadioptric imaging devices, image measurements are represented as 3D bearing vectors: a unit vector originating at the camera center and pointing toward the landmark. Each bearing vector has only two degrees of freedom, which are the azimuth and elevation inside the camera reference frame as formulated in [19]. Because a bearing vector has only two degrees of freedom, we frequently refer to it as a 2D information and it is normally expressed in a camera reference frame.

The disambiguation step that has, finally, to be performed by each thread consists in calculating the sum of reprojection error of the triangulated 3D points relative to the corresponding bearing vectors used to compute the model. Finally,

a single 4×3 transformation into the world reference frame matrix is returned by each thread referring to the lowest score of reprojection error between all essential matrices and pose configurations (up to 40). The transformation matrix is directly obtained from the already calculated rotation and translation.

Indeed, the triangulation method used in our implementation follows the general scheme employed in [19]. The reprojection error of 3D bearing vectors was proposed in [19] as well, and is computed by considering the angle between the measured bearing vector $\mathrm{f_{meas}}$ and the reprojected one $\mathrm{f_{repr}}$. In fact, the scalar product of $\mathrm{f_{meas}}$ and $\mathrm{f_{repr}}$ directly gives the angle between them, which is equal to $\cos\theta$ as illustrated in Figure 3. The reprojection error is, therefore, expressed as

$$\epsilon = 1 - f_{meas}^{T} f_{repro} = 1 - \cos\theta. \tag{19}$$

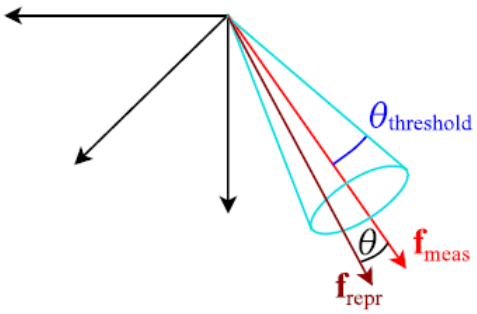


Fig. 3. Reprojection error computation in Opengv (Source: [19]).

RANSAC Rating Kernel In order to validate each estimated model, we compute a loss value for each datapoint of the dataset. The loss value is used to verify the model by computing the reprojection error of all triangulated bearing vectors of the dataset. Outliers are subsequently found by thresholding the reprojection errors, and the best model refers to the one with the maximum number of inliers. As the entire operation is in 3D, we use the thresholding scheme adopted in the Opengv library [19]. This latter uses a threshold angle $\theta_{threshold}$ to constrain f_{repr} to lie within a cone of axis f_{meas} and of opening angle $\theta_{threshold}$ as depicted in Figure 3. The threshold error is given by

$$\epsilon_{threshold} = 1 - \cos\theta_{threshold} = 1 - \cos(arctan\frac{\psi}{l}), \tag{20}$$

where ψ refers to the classical reprojection error threshold expressed in pixels and l to the focal length.

The model validation process considers multiple accesses to global memory to evaluate whether each correspondence of the dataset is an inlier or an outlier which is a very time consuming. The shared memory is, hence, used as a cache to accelerate computations. The RANSAC rating kernel employs a block level parallelism and is lauched with *iterations* blocks to make each block handle a RANSAC model and $8 \times warpsize$ threads. Since $warpsize = 32$, a total of 256 threads are launched per block and each thread in the block evaluates a point. To load datapoints in shared memory, a buffer is allocated of size $256 \times s$ where s refers to the size of each datapoint. In case of bearing vectors, $s = 6$. Each thread triangulates bearing vector correspondances into a 3D point and computes its reprojection error according to Equation 19. This latter is, thereafter, compared to the precalculated threshold according to Equation 20 to decide whether the correspondance refers to an inlier or to an outlier. In our implementation, the number of inliers for 256 values is automatically returned via:

```
inlier_count=__syncthreads_count(
                reproj_error[threadIdx.x]<threshold);
```

The process of loading data into buffer and evaluating 256 reprojection errors is repeated ceil(datasetCount/256) times.

RANSAC Best Model Computation Kernel This kernel is launched with one block and *itearations* threads and performs a reduction in shared memory to derive the best model which refers to the one with the maximum number of inliers.

3.2 CUDA-based Relative Motion Estimation from Seven Correspondences

The implementation of the seven-point method relies on many common components with the five-point method presented above. Our parallelization strategy relies on performing the RANSAC iterations in parallel, as well. This suggests redesigning the sequential code as has been done for the five point case. In this section, we will only present the key steps that are different from the previous implementation. We recall that, as has been stated before, the fundamental matrix has 7 degrees of freedom and requires seven correspondences to be estimated. As we are working with normalized bearing vectors on the unit sphere, our data is, therefore, normalized, and we directly obtain the essential matrix. As has been presented in Section 2.2, the algorithm requires the computation of the null-space of the constraint matrix formed from 7 correspondences randomly chosen. The null-space is two-dimensional resulting in two independant vectors. These latters lead to a 3^{rd} degree polynomial that has at most 3 real root representing the possible fundamental matrices solutions (or the essential matrices in the calibrated case). This requires the rcovery of the null-space and the resolution of a 3^{rd} degree polynomial.

Null-space recovery The constraint matrix, constructed from 7 correspondences is of size 7×9. The null-space recovery is, generally, performed by SVD. We use instead the same method presented in Section 3.1 based on a QR decomposition and use as well the batched interface of the CuBLAS library to simultaneously find the null-spaces of a batch of small matrices, each referring to a RANSAC iteration.

Cubic Polynomial Root Solver In the 7-point algorithm, finding the fundamental matrices solutions or the essential matrices for the calibrated case is reduced to solving a 3^{rd} degree polynomial in the form of Equation 10. Several state of the art methods can be used to solve a general polynomial including Durand-kerner method presented in Section 3.1. However, as we are dealing with only a cubic degree, we have opted for a direct solver, namely Cardano's Formula [22].

Robust Pose estimation on GPU The advantage of the GPU is gained through the execution of the RANSAC iterations in parallel. The RANSAC estimation process consists in two major steps executed by two separate kernels namely, hypotheses computation kernel and loss values computation kernel.

Hypotheses computation kernel This kernel is executed on the CUDA device with a given number of CUDA threads T and CUDA blocks B depending on the number of required iterations as $N = T.B$ We take in general the number of threads T as multiple of warp size. This kernel consists of the steps explained below.

1. At the beginning of each iteration, each thread performs the computation of 7 random numbers that are used to choose the subset for further steps. In our implementation, Marsaglia's random numbers generator has been used to generate random numbers on GPU [23].
2. The second step consists in simultaneously computing the different hypotheses including finding up to 3 possible essential matrices, solutions of the 3^{rd} degree polynomial. Each solution is decomposed, then, into rotation R and translation t and a disambiguation is ,subsequently, performed to pick up the right solution. At the end of this stage, transformation matrices (with the form $[R|t]$) are contiguously written in the device's global memory at the number of RANSAC iterations.

RANSAC Rating Kernel In order to validate each estimated hypothesis obtained on exit of the hypotheses computation kernel, we compute a loss value for each datapoint of the dataset. The loss value is used to verify the hypothesis by computing the reprojection error of all triangulated bearing vectors of the dataset as presented in Section 3.1. Outliers are subsequently found by thresholding the reprojection errors, and the best model refers to the hypothesis with the maximum number of inliers.

An additional kernel is, finally, used aiming at determing the best model which refers to the one that has the maximum number of inliers by performing a reduction in shared memory.

3.3 CUDA-based Relative Motion Estimation from Eight Correspondences

The egomotion estimation using the eight-point algorithm on CUDA follows the same scheme presented in the previous section dealing with seven correspondences. The null-space is obtained via a QR deomposition as has been previously discussed and is one dimensional which gives one essential matrix solution. The implementation mainly involves two kernels, the first for hypotheses computation that is launched with T threads and B blocks to simultaneously compute N RANSAC iterations where $N = T.B$. The second is for rating the RANSAC hypotheses ans is launched with N blocks and 256 threads and involves the bearing vectors rating scheme. Finally, an additional kernel is used to determine the best model which refers to the one that has the maximum number of inliers. However, to avoid the costly iterative SVD, the null-space is determined via QR instead of SVD and the rank 2 constraint that has to be enforced according to Equation 6 is directly implemented using an SVD computation for 3×3 matrices.

4 Results

In this section we evaluate the speed and accuracy of our CUDA based essential matrix solver within RANSAC and compare it against the CPU based implementation for general relative camera motion provided in the OpenGV library. This latter is an *openSource* library that operates directly in 3D and provides implementations to solve the problems of computing the absolute or relative pose of a generalized camera [19].

4.1 Random Problem Generation

To make synthetic data for our tests, we used the automatic benchmark for relative pose included in the Matlab interface of the OpenGV library. We used the provided experiment to create a random relative pose problem, that is, correspondences of bearing vectors in two viewpoints using two cameras at the number of 1000 correspondences. In fact, the number of 1000 correspondences has been chosen based on an averaged number obtained from real images. The experiment returns the observations in two viewpoints plus the ground truth values for the relative transformation parameters.

4.2 Timing

We have measured the mean time while running on the GPU and CPU (using OpenGV library). To compute the mean time, each estimation is repeated

20 times. The repetition rate is required since a single estimation can be much slower or much faster than the mean due to the randomization. We only present results of computations for single-precision datatype as the precision loss due to single precision dosen't really affect the localization estimation in a real scenario. In addition, visual odometry is generally used with an optimization process to reduce the drift caused by the run-time accumulation error. The system on which the code has been evaluated is equipped with an i7 CPU running at up to 3.5 GHz, the intel i7 CORE. The CUDA device is an NVIDIA GeForce GTX 850M running at 876 MHz with 4096 MB of GDDR5 device memory. The evaluation has been performed with CUDA version 7.5 integrated with VisualStudio 2012. At the first execution of the estimation, memory allocations have to be performed. This is required only once and takes about 6 ms. To evaluate our implementation, 10 outlier ratios from $\epsilon = 0.05$ to $\epsilon = 0.5$ in steps of $\epsilon = 0.05$ are evaluated. Figure 4 shows the number of required RANSAC iterations for the essential matrix estimation from 5, 7 and 8 correspondences for the 10 outlier ratios.

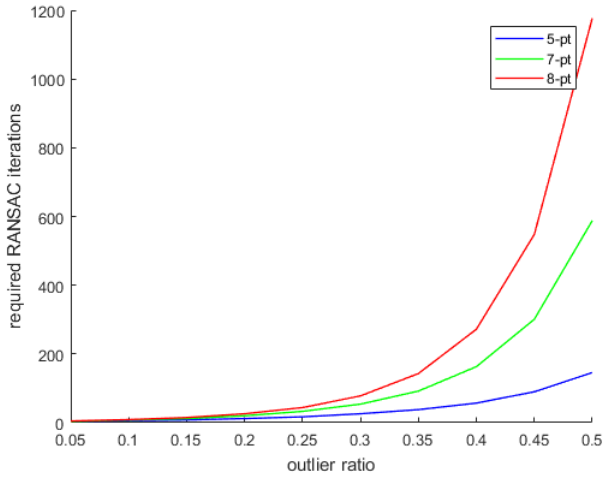


Fig. 4. Required RANSAC iterations vs outlier ratio.

In Figure 5, we show the performance results of estimating camera relative pose from sets of five 2D bearing vectors correspondences. Firstly, in Figure 5(a), we compare the mean computation time of CPU and GPU implementations, in single precision. We show a mean computation time even more important for CPU reaching 86 ms for an outlier ratio $\epsilon = 0.5$ against 18 ms for GPU. With an outlier ratio of $\epsilon = 0.5$ which is common for the essential matrix estimation from automatically computed point correspondences, we show in Figure 5(b) that the

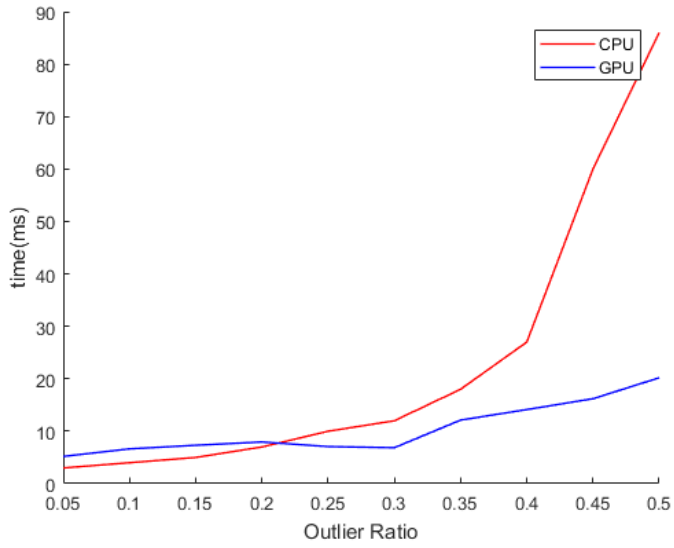
speedup is above 4× compared to the CPU implementation. Furthermore, it is useful to visualize the intersection between each CPU and GPU evaluation, *i.e.* the outlier ratio where the speedup is equal to one. Figure 5(b) shows that there is no speedup for lower outlier ratios $\epsilon \leq 0.2$. This is because the needed number of iterations for $\epsilon = 0.2$ is only 12 iterations. However, the minimum number of iterations used in GPU based implementation is 32 iterations referring to the warp size.

In Figure 6, we show the performance results of estimating camera relative pose from sets of seven 2D bearing vectors correspondences. In Figure 6(a), we compare the mean computation time of CPU and GPU implementations, in single precision. We show a mean computation time for CPU reaching 266 ms for an outlier ratio $\epsilon = 0.5$ against 3.5 ms for GPU allowing a highly important speedup of 76× as depicted in Figures 6(a) and 6(b) for mean time in ms and speedup respectively.

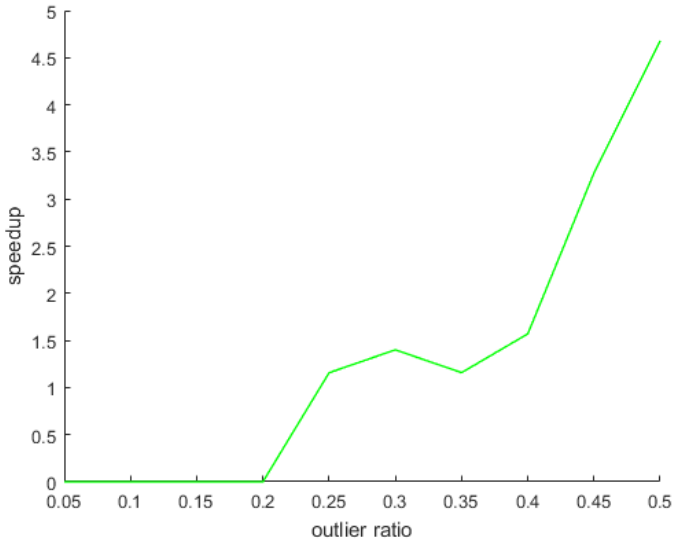
In Figure 7, we show the performance results of estimating camera relative pose from sets of eight 2D bearing vectors correspondences. As shown in Figure 7(a), a speedup of almost 57× is achieved for an outlier ratio of $\epsilon = 0.5$. In fact, the seven point scheme achieves more speedup as for $\epsilon = 0.5$, 588 iterations are required to attain a probability of 0.99 that the subset is outlier-free against 1177 iterations when relying on eight correspondences.
In a second series of experiments, a fixed number of RANSAC iterations equal to 1024 is used to evaluate the performance of the three egomotion estimation schemes. In fact, a high number of iterations is sometimes needed, priorily determined, in order to estimate the covariance of the data as for instance in [24]. In Figure 8, we evaluate the time in ms and the speedup of our CUDA-based implementations of essential matrix estimation within RANSAC from five, seven and eight correspondences. Due to the implementations complexities, we only launch 256 parallel threads in the case of *5-pt algorithm* and the kernels are therefore launched 4 times serially by the CPU to perform the 1024 iterations. In the case of the *7-pt algorithm* and *8-pt algorithm*, 512 threads are issued in parallel and the kernels are serially launched 2 times. Figure 8(a) shows an even more important CPU time reaching 464 ms, 303 ms and 269 ms for the *5-pt algorithm*, *7-pt algorithm* and the *8-pt algorithm* respectively against 50.34 ms, 6.31 ms and 5.5 ms for the CUDA-based implementations. This allows to achieve almost 9× speedup for the *5-pt algorithm* and almost 48× speedup for both the *7-pt algorithm* and *8-pt algorithm*.

5 Conclusion

In this paper we presented a CUDA-accelerated 2D-2D feature-based egomotion estimation from five, seven and eight correspondences. Feature-based egomotion is typically used within RANSAC in order to deal with erroneous feature correspondences known as outliers. We presented our parallelization strategy, based mainly on performing the required RANSAC iterations in parallel on the

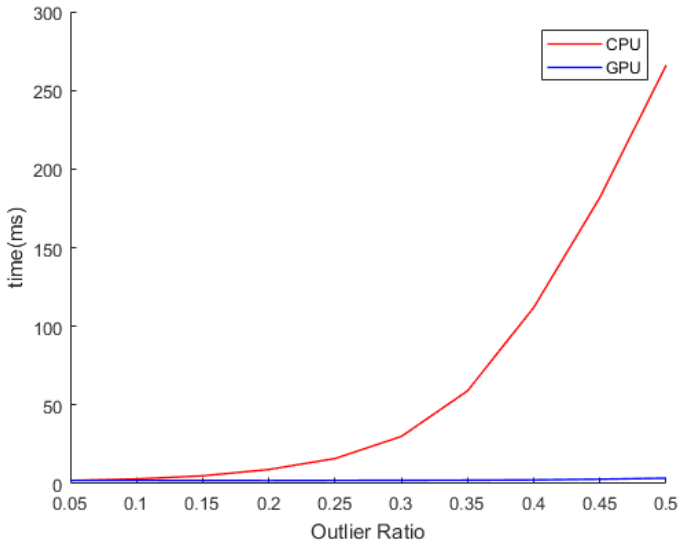


(a) timing

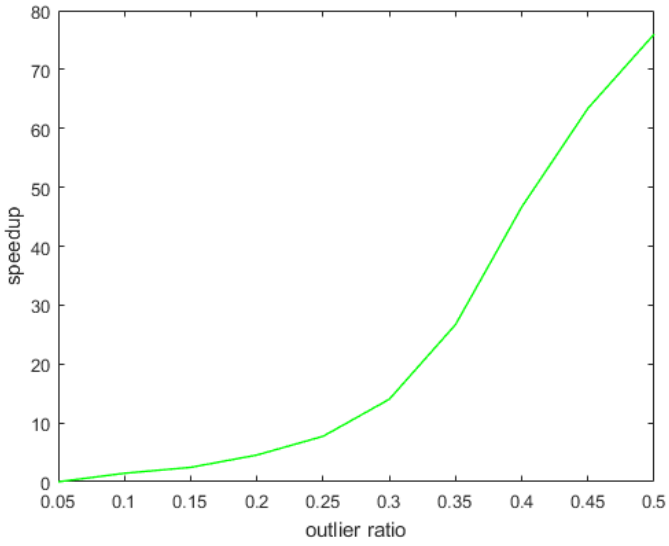


(b) speedup

Fig. 5. Performance of essential matrix estimation with RANSAC from 5 correspondences.

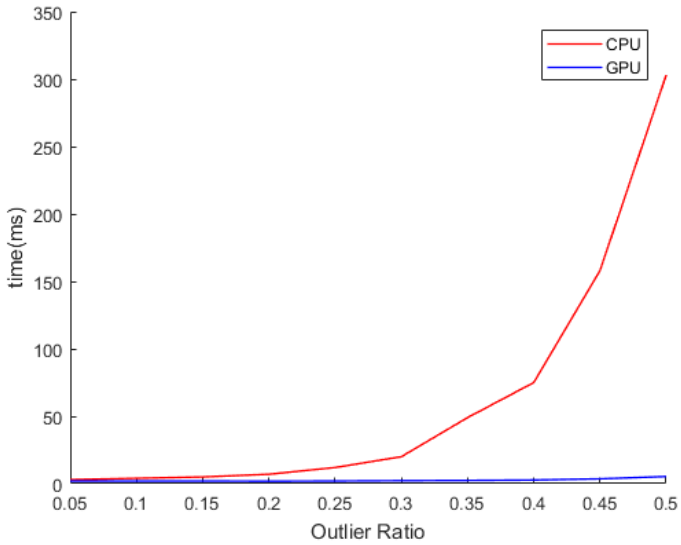


(a) timing

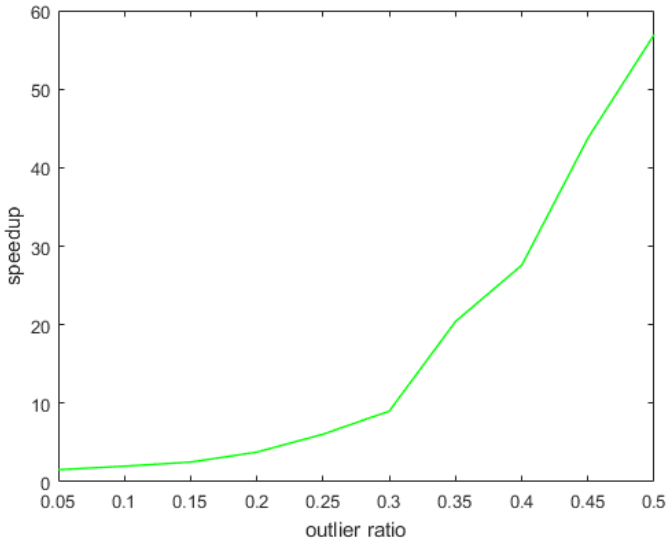


(b) speedup

Fig. 6. Performance of essential matrix estimation with RANSAC from 7 correspondences.

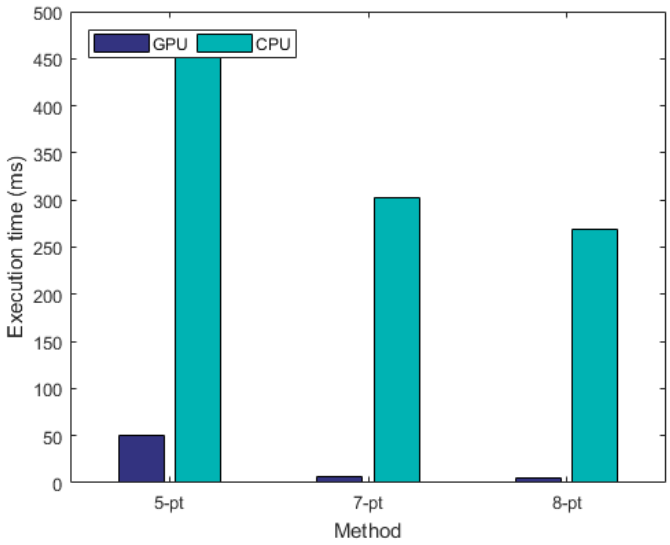


(a) timing

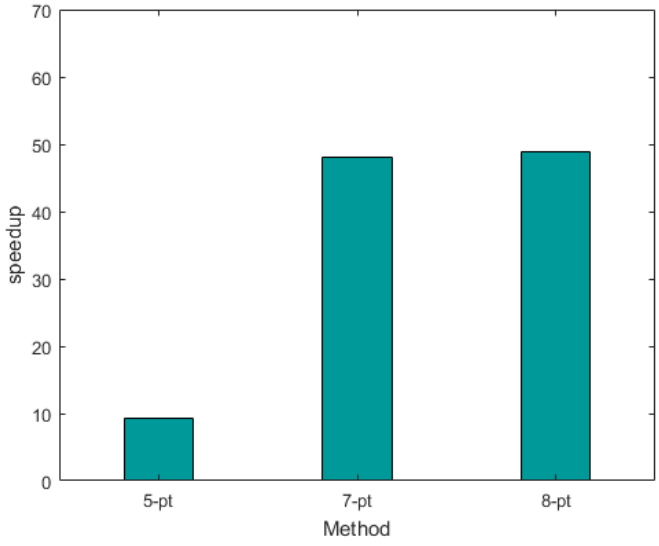


(b) speedup

Fig. 7. Performance of essential matrix estimation with RANSAC from 8 correspondences.



(a) timing



(b) speedup

Fig. 8. Performance of essential matrix estimation with RANSAC from 5, 7 and 8 correspondences.

CUDA GPU. We, hence, designed a mapping of the five-point essential matrix using Gröbner basis to CUDA ressources and programming model as well as the seven-point and eight-point schemes. Our hardware-specific implementations dealt with multiple CUDA features such as the the batched interface of the cuBLAS library and the dynamic parallelism. In addition, in order to deal with all central camera models including perspective, dioptric, omnidirectional and catadioptric imaging devices, we used a novel scheme based on representing feature measurements as bearing vectors. This representaion suggested a specific rating measure for RANSAC which is based on the computation of the reprojection error of triangulated 3D points from corresponding bearing vectors. An evaluation of our implementation was presented and the mean computation time of RANSAC for different outlier ratios was measured. For an outlier ratio $\epsilon = 0.5$, common for the essential matrix estimation from automatically computed point correspondences, a speedup of 4 times faster than the CPU counterpart was achieved for the five-point version. Higher speedups were shown for the seven-point and the eight-point versions reaching 76 times and 57 times respectively. The five-point version is known to have a better accuracy at the expense of complexity, whereas the faster seven and eight point versions are preferred when integrating an optimization process aiming at reducing the accumulated run-time error.

References

1. Thrun, S., Leonard, J.J.: Simultaneous 37. Simultaneous Localization and Mapping. Springer Handbook of Robotics. 19 (2008).
2. Hager, G., Hebert, M., Hutchinson, S.: Editorial: Special issue on vision and robotics, parts i and II. International Journal of Computer Vision. 74, 217218 (2007).
3. Neira, J., Davison, A.J., Leonard, J.: Guest Editorial Special Issue on Visual SLAM. 24, 929931 (2008).
4. Davison, A.J., Reid, I.D., Molton, N.D., Stasse, O.: MonoSLAM: Real-Time Single Camera SLAM. IEEE Transactions on Pattern Analysis and Machine Intelligence. 29, 10521067 (2007).
5. Williams, B., Klein, G., Reid, I.: Real-time SLAM relocalisation. In: Proc. International Conference on Computer Vision (2007).
6. Newcombe, R.A., Lovegrove, S.J., Davison, A.J.: DTAM: Dense tracking and mapping in real-time. Presented at the November (2011).
7. Scaramuzza, D., Fraundorfer, F.: Visual Odometry [Tutorial]. IEEE Robotics & Automation Magazine. 18, 8092 (2011).
8. Fraundorfer, F., Scaramuzza, D.: Visual Odometry: Part II: Matching, Robustness, Optimization, and Applications. IEEE Robotics & Automation Magazine. 19, 7890 (2012).
9. Nister, D.: An efficient solution to the five-point relative pose problem. IEEE Transactions on Pattern Analysis and Machine Intelligence. 26, 756770 (2004).
10. Stewnius, H., Engels, C., Nistr, D.: Recent developments on direct relative orientation. ISPRS Journal of Photogrammetry and Remote Sensing. 60, 284294 (2006).
11. Hartley, R., Zisserman, A.: Multiple View Geometry in Computer Vision, Second Edition. Cambridge Univ. Press, Cambridge U.K.

12. Fischler, M.A., Bolles, R.C.: Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. Commun. ACM. 24, 381395 (1981).
13. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro. 28, 3955 (2008).
14. Li, B., Zhang, X., Sato, M.: Pitch angle estimation using a Vehicle-Mounted monocular camera for range measurement. In: 2014 12th International Conference on Signal Processing (ICSP). pp. 11611168 (2014).
15. Yonglong, Z., Kuizhi, M., Xiang, J., Peixiang, D.: Parallelization and Optimization of SIFT on GPU Using CUDA. In: 2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing. pp. 13511358 (2013).
16. Wu, C., Agarwal, S., Curless, B., Seitz, S.M.: Multicore bundle adjustment. In: CVPR 2011. pp. 30573064 (2011).
17. Chang, C.-C., Lin, C.-J.: LIBSVM: A Library for Support Vector Machines. ACM Trans. Intell. Syst. Technol. 2, 27:127:27 (2011).
18. NVIDIA documentation, cuBLAS, http://docs.nvidia.com/cuda/cublas/index.html.
19. Kneip, L., Furgale, P.: OpenGV: A unified and generalized approach to real-time calibrated geometric vision. In: 2014 IEEE International Conference on Robotics and Automation (ICRA). pp. 18 (2014).
20. Ouerghi, S., Boutteau, R., Savatier, X., Tlili, F.: CUDA Accelerated Visual Egomotion Estimation for Robotic Navigation. In Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 4: VISAPP, 107-114, (2017).
21. Stewenius, H. and Engels, C. Matlab code for solving the fivepoint problem. `http://vis.uky.edu/œstewe/FIVEPOINT/`. Online (2008).
22. Cardano formula - Encyclopedia of Mathematics, `https://www.encyclopediaofmath.org/index.php/Cardano_formula`. Online.
23. Marsaglia, G.: Random Number Generators. Journal of Modern Applied Statistical Methods. 2, (2003).
24. Ouerghi, S., Boutteau, R., Savatier, X., Tlili, F.: Visual Odometry and Place Recognition Fusion for Vehicle Position Tracking in Urban Environments. Sensors. 18, 939 (2018).