CUDA Accelerated Visual Egomotion Estimation for Robotic Navigation

Safa Ouerghi, Rémi Boutteau, Xavier Savatier, Fethi Tlili

CUDA Accelerated Visual Egomotion Estimation for Robotic Navigation

Safa Ouerghi[1], Remi Boutteau[2], Xavier Savatier[2] and Fethi Tlili[1]
[1]*Carthage Univ, Sup'Com, GRESCOM, 2083 El Ghazela, Tunisia*
[2]*Normandie Univ, UNIROUEN, ESIGELEC, IRSEEM, 76000 Rouen, France*
{*safa.ouerghi, fethi.tlili*}*@supcom.tn*, {*Remi.Boutteau, Xavier.Savatier*}*@esigelec.fr*

Keywords: Egomotion, structure from motion, robotics, CUDA, GPU.

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 correspondances have been proposed. Five-point methods represent the minimal number of required correspondences to estimate the essential matrix, raised special interest for their application in a hypothesize-and-test framework. 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 performed using the Gröbner basis version of 5-point algorithm, complemented with robust estimation. The description of the hardware-specific implementation considerations as well as the parallelization methods employed are given in detail. Performance analysis against existing CPU implementation is also given, showing a speedup 4 times faster than the CPU for an outlier ratio $\varepsilon = 0.5$, common for the essential matrix estimation from automatically computed point correspondences. More speedup was shown when dealing with higher outlier ratios.

1 INTRODUCTION

Accurate localization is a fundamental issue in autonomous navigation that has been extensively studied by the Robotics community. During the last years, cameras have become very popoular in Robotics allowing the developement of several vision-based methods for a real time localization. These methods primarily follow two main paradigms namely, SLAM (Simultaneous localization and Mapping)(Durrant-Whyte and Bailey, 2006; Dissanayake et al., 2001) and real-time SFM (Structure from Motion) or Visual Odometry (Nister et al., 2006; Maimone et al., 2007; I. Comport et al., 2010). While SLAM methods tackle the issue of concurrent localization and mapping of a vehicle in an unknown environment, visual odometry calculates the egomotion by incrementally estimating the rotation and translation undergone by the vehicle using only the input of a single or multiple cameras. The visual odometry pipeline for the stereo scheme consists mainly in finding corresponding features between adjacent images 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 (Nister, 2004) 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 (Stewenius et al., 2006). 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 RANSAC and has been first proposed by Fischler and Bolles (Fischler and Bolles, 1981). 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. 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 (Lindholm et al., 2008). In fact, recently, many problems are being solved using programmable graphics hardware including feature matching and triangulation (Li et al., 2014), feature detectors (Yonglong et al., 2013), large non-linear optimization problems such as bundle adjustment (Wu et al., 2011) and learning algorithms (Chang and Lin, 2011).

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 matrix estimation in section 2. Section 3 details the CUDA based implementation of the fivepoint essential matrix estimation algorithm within RANSAC. 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 BACKGROUNDS

In this section, we provide an overview of the important backgrounds underlying the robust essential matrix estimation from 5 correspondances.

2.1 Essential matrix

The essential matrix E is a 3×3 matrix expressing the epipolar geometry between two calibrated camera systems (Hartley and Zisserman, 2004). That is, if a 3D point X is viewed in two images at locations u and v (where u and v are calibrated homogeneous image coordinates), then u and v are related such that

$$v^T E u = 0. \tag{1}$$

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

$$E = [t]_\times R, \tag{2}$$

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. Therefore, E is only determined up to a scale.

Expanding Equation 1 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

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

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

$$EE^T E - \frac{1}{2} trace(EE^T) E = 0. \tag{3}$$

2.2 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 (Hartley and Zisserman, 2004). 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 (Nister, 2004), and Stewenius et al. (Stewenius et al., 2006) both work by solving these nine equations.

Stewenius et al. first showed that the equations can be written as

$$MX = 0,$$

where M is a 10×20 matrix.

After gauss-jordan elimination, the system can be written

$$[I\, B]X = 0,$$

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 (Nister, 2004), 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

$$D = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{4}$$

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

$$N = \frac{\log(1-p)}{\log(1-(1-\varepsilon)^k)} \tag{5}$$

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

$$\varepsilon = 1 - \frac{\gamma}{n}, \tag{6}$$

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

3 GPU IMPLEMENTATION OF RELATIVE POSE FROM FIVE 2D-2D CORRESPONDENCES

In this section, we present the implementation details of the essential matrix estimation from five correspondences within RANSAC using the CUDA programming model. As stated before, the eigenvalues of the action matrix contain the essential matrices solutions according to Stewenius's method (Stewenius et al., 2006). However, a device based eigenvalue computation on CUDA doesn't exist yet. Hence, we have relied on the Matlab code provided by Chris Engels, based on the reduction to a single polynomial (Stewenius and Engels, 2008). This is done through the computation of the action matrix characteristic polynomial roots, equivalent to the action matrix eigenvalues.

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. Furthermore, we have relied on the use of cuBLAS, a high-performance implementation of BLAS-3 routines, for linear algebra computations (NVIDIA, 2015). As the matrices sizes in our problem are below 32×32, we have particularly exploited the batched interface of the cuBLAS libraiary where many small dense matrices factorizations, to be performed simultaneously, are provided.

Thus, 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.

3.1 Cublas based kernel

This kernel is launched with one block and a number of threads equal to the number of required RANSAC iterations. The high level interface exposed by all implementations in this kernel is cuBLAS batched interface for solving a batch of N different systems with double precision. 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 identical 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();
```

Figure 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 carriying 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 $(PM = LU)$.
2. Permutation of the array b with the array of pivots P $(y = Pb)$.
3. Solution of the triangular lower system $(Lc = y)$.
4. Solution of the upper system to obtain the final solution $(Ux = c)$

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 *cublas-DgetrsBatched,* solution of the system $MX = 0$.

3.2 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 covolution 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.

3.2.1 Durand-Kerner Method

The Durand_Kerner Method allows the extraction of all roots $\omega_1, ..., \omega_n$ of a polynomial

$$p(z) = \sum_{i=0}^{n} a_i z^{n-i}, \tag{7}$$

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

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:

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

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.

3.2.2 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 7 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;
  }
```

Figure 2: Pseudocode of batched Durand-Kerner method on CUDA.

As explicited in Subsection 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 3, that is proposed in (Nister, 2004) (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 (Kneip and Furgale, 2014). 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 trinagulated 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 (Kneip and Furgale, 2014). The reprojection error of 3D bearing vectors was proposed in (Kneip and Furgale, 2014) as well, and is computed by considering the angle between the measured bearing vector f_{meas}, and the reprojected one f_{repr}. In fact, the scalar product of f_{mes} and 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

$$\varepsilon = 1 - f_{meas}^T f_{repro} = 1 - \cos\theta. \tag{8}$$

3.3 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

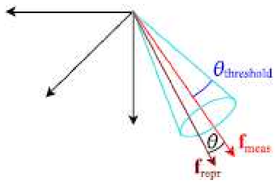


Figure 3: Reprojection error computation in Opengv (Source: (Kneip and Furgale, 2014)).

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 librairy (Kneip and Furgale, 2014). 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

$$\varepsilon_{threshold} = 1 - \cos\theta_{threshold} = 1 - \cos(arctan\frac{\psi}{l}), \tag{9}$$

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 correspondance of the dataset is an inlier or an outlier which is a very execution-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 handles a RANSAC model and $8 \times warpsize$ threads. Since $warpsize = 32$, a total of 256 threads is 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 8. This latter is, thereafter, compared to the precalculated threshold according to Equation 9 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.

3.4 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.

4 EVALUATION

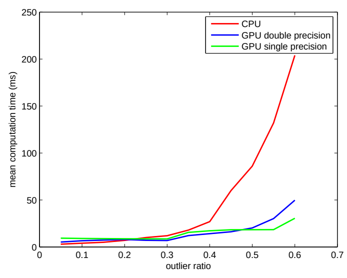
In this section we evaluate the speed and accuracy of our CUDA based essential matrix solver within RANSAC and compare it against the CPU based implementation for general relative camera motion provided in the OpenGV library. This latter is an *open-Source* librairy that operates directly in 3D and provides implementations to solve the problems of computing the absolute or relative pose of a generalized camera (Kneip and Furgale, 2014).

4.1 Random Problem Generation

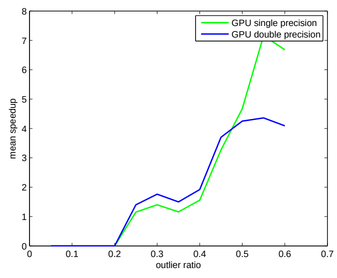
To make synthetic data for our tests, we used the automatic benchmark for relative pose included in the Matlab interface of the OpenGV librairy. We used the provided experiment to create a random relative pose problem, that is, correspondences of bearing vectors in two viewpoints using two cameras at 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 both viewpoints (assumed to be a stereo camera system), 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 librairy). 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 will present results of computations for both single-precision and double-precision datatypes. 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 $6ms$. To evaluate our implementation, 12 outlier ratios from $\varepsilon = 0.05$ to $\varepsilon = 0.6$ in steps of $\varepsilon = 0.05$ are evaluated. In Figure 4, we show the performance results of estimating camera relative pose from sets of 2D bearing vectors correspondences. Firstly, in Figure 4a, we compare the mean computation time of CPU and GPU implementations, in single and double precision. We show a mean computation time even more important for CPU reaching $86ms$ for an outlier ratio $\varepsilon = 0.5$ against $20.2ms$ for GPU in double precision and $18ms$ in single precision. With an outlier ratio of $\varepsilon = 0.5$ which is common for the essential matrix estimation from automatically computed point correspondences, we show in figure 4b that the speedup is about $4\times$ compared to the CPU implementation in single and double precision. The speedup becomes more important for higher outlier ratios reaching $7\times$ in single precision for $\varepsilon = 0.6$ against $4\times$ in double precision. 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 4 shows that there is no speedup for lower outlier ratios $\varepsilon \leq 0.2$. This is because the needed number of iterations for $\varepsilon = 0.2$ is only 12 iterations. However, the minimum number of iterations used in GPU based implementation is 32 iterations referring to the warp size.



(a) mean time



(b) speedup

Figure 4: Performance of essential matrix estimation with RANSAC.

4.3 Accuracy

In Figure 5, we present the Root-mean-square error (RMSE) between the ground truth rotation matrix and our Cuda based implementation rotation matrix of the best model for different outlier ratios, for both single and double precision datatypes. Overall, single precision datatype showed good performance while ensuring higher levels of speedup for higher outlier ratios. The accuracy loss is mostly due to the batched durand-kerner version for solving 10^{th} degree polynomials where the maximum number of iterations is fixed.

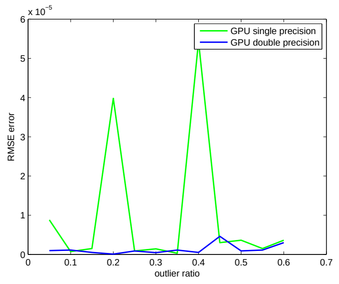


Figure 5: RMSE in rotation between ground truth and CUDA best model.

5 CONCLUSIONS

In this paper we have presented a 2D-2D robust motion estimation on CUDA which is applicable to a wide range of problems and especially to autonomous navigation. We presented our parallelization strategy, based mainly on performing the required RANSAC iterations in parallel. We described our implementation dealing with several levels of parallelism namely, warp level parallelism, block level parallelism and thread level parallelism. In addition, we adapted the five-point essential matrix using Gröbner basis to CUDA ressources and programming model. Furthermore, we described our RANSAC implementation and the rating measure used which is based on the computation of the reprojection error of triangulated 3D points from bearing vectors. An evaluation of our implementation has been presented and the mean computation time of RANSAC for different outlier ratios has been measured. Overall, the implementation showed good performance, and a speedup 4 times faster than the CPU was measured for an outlier ratio $\varepsilon = 0.5$, common for the essential matrix estimation from automatically computed point correspondences. More speedup was shown when dealing with higher outlier ratios.

REFERENCES

Chang, C.-C. and Lin, C.-J. (2011). Libsvm: A library for support vector machines. volume 2, pages 1–27.

Dissanayake, M. W. M. G., Newman, P., Clark, S., Durrant-whyte, H. F., and Csorba, M. (2001). A solution to the simultaneous localization and map building (slam) problem. In *IEEE Transactions on Robotics and Automation*, volume 17, pages 229–241.

Durrant-Whyte, H. and Bailey, T. (2006). Simultaneous localisation and mapping (slam): Part i the essential algorithms. volume 2, page 2006.

Fischler, M. A. and Bolles, R. C. (1981). Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. volume 24, pages 381–395.

Hartley, R. I. and Zisserman, A. (2004). *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition.

I. Comport, A., Malis, E., and Rives, P. (2010). Real-time quadrifocal visual odometry. volume 29, pages 245–266.

Kneip, L. and Furgale, P. (2014). Opengv: A unified and generalized approach to real-time calibrated geometric vision.

Li, B., Zhang, X., and Sato, M. (2014). Pitch angle estimation using a vehicle-mounted monocular camera for range measurement. volume 28, pages 1161–1168.

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). Nvidia tesla: A unified graphics and computing architecture. volume 28, pages 39–55.

Maimone, M., Cheng, Y., and Matthies, L. (2007). Two years of visual odometry on the mars exploration rovers. volume 24, page 2007.

Nister, D. (2004). An efficient solution to the five-point relative pose problem. volume 26, pages 756–777.

Nister, D., Naroditsky, O., and Bergen, J. (2006). Visual odometry for ground vehicle applications. volume 23, page 2006.

NVIDIA (2015). Cublas documentation. http://docs.nvidia.com/cuda/cublas/. Online.

Stewenius, D. H., Engels, C., and Nistr, D. D. (2006). Recent developments on direct relative orientation. volume 60, pages 284–294.

Stewenius, H. and Engels, C. (2008). Matlab code for solving the fivepoint problem. http://vis.uky.edu/~stewe/FIVEPOINT/. Online.

Wu, C., Agarwal, S., Curless, B., and Seitz, S. (2011). Multicore bundle adjustment. pages 3057–3064.

Yonglong, Z., Kuizhi, M., Xiang, J., and Peixiang, D. (2013). Parallelization and optimization of sift on gpu using cuda. pages 1351–1358.