Scientific Research Publishing

Journal of Computer and Communications, 2016, 4, 1-17
http://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Acceleration of Points to Convex Region Correspondence Pose Estimation Algorithm on GPUs for Real-Time Applications

Raghu Raj P. Kumar, Suresh S. Muknahallipatna, John E. McInroy

Department of Electrical and Computer Engineering, University of Wyoming, Laramie, USA
Email: raghuraj19@gmail.com, sureshm@uwyo.edu, mcinroy@uwyo.edu

How to cite this paper: Kumar, R.R.P., Muknahallipatna, S.S. and McInroy, J.E. (2016) Acceleration of Points to Convex Region Correspondence Pose Estimation Algorithm on GPUs for Real-Time Applications. *Journal of Computer and Communications*, **4**, 1-17.
http://dx.doi.org/10.4236/jcc.2016.417001

Received: October 31, 2016
Accepted: December 26, 2016
Published: December 29, 2016

Copyright © 2016 by authors and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

In our previous work, a novel algorithm to perform robust pose estimation was presented. The pose was estimated using points on the object to regions on image correspondence. The laboratory experiments conducted in the previous work showed that the accuracy of the estimated pose was over 99% for position and 84% for orientation estimations respectively. However, for larger objects, the algorithm requires a high number of points to achieve the same accuracy. The requirement of higher number of points makes the algorithm, computationally intensive resulting in the algorithm infeasible for real-time computer vision applications. In this paper, the algorithm is parallelized to run on NVIDIA GPUs. The results indicate that even for objects having more than 2000 points, the algorithm can estimate the pose in real time for each frame of high-resolution videos.

Keywords

Pose Estimation, Parallel Computing, GPU, CUDA, Real Time Image Processing

1. Introduction

In our previous work, a novel pose estimation algorithm based on points to region correspondence was proposed in [1]. Given the points on an object and the convex regions in which the correspondent image points lie, the concrete values of position and orientation between the object and the camera are found based on points to regions correspondence. The unit quaternion representation of rotation matrix and convex Linear Matrix Inequalities (LMI) optimization me-

thods are used to estimate the pose. By loosening the requirement of precise point-to-point correspondence and using convex LMI formulations, the algorithm provided a more robust pose estimation method. While the pose estimation experiments yielded satisfactory results, the test cases considered were small objects with four points per object. While this may be reasonable for lab experiments, with the current high and ultra-high resolution images, larger objects are captured on images. Having a low number of points for larger objects, the pose estimation algorithm will have a low accuracy, as shown in [1]. It is also shown in [1] that for more points on a given object, the accuracy of the pose estimation increases. Moreover, some pose estimation based video processing applications require processing of larger objects in high-resolution images at high frame rates (300) in real-time, *i.e.*, 300 high-resolution images have to be processed for an object with large number of points within a second to obtain the pose. Since, the serial execution time of the pose estimation algorithm is bound to the number of points, *i.e.*, with a higher number of points, the execution time of the pose estimation algorithm also increases. This makes the pose estimation involving large objects with high accuracy, infeasible for real-time applications. Therefore, the algorithm requires further analysis to make it suitable for real-time applications.

The pose estimation algorithm, described in [1], is an iterative 2D search algorithm. The number of computations for the algorithm is dependent on two factors: a) number of points on objects, also known as markers and b) accuracy of the estimated pose. The number of computations within a single iteration is determined by the number of markers, *i.e.*, the number of computations within each iteration increases linearly with an increase in the number of markers. The number of iterations is determined by the desired accuracy, *i.e.*, the number of iterations increases non-linearly for higher order accuracy, as it is a 2D search. However, the iterations being independent of one another make concurrent execution of all iterations feasible. Since the number of concurrent iterations grows non-linearly with an increase in the desired accuracy, more execution cores are needed to maximize concurrency. Thus, the algorithm is well suited for General Purpose Computation on GPU (GPGPU) parallelization.

In this paper, we propose a GPGPU parallelization approach for the pose estimation algorithm. The goal is to achieve optimal and scalable implementation of the pose estimation algorithm on GPUs, making it suitable for real-time applications. To achieve our goal, we parallelize the algorithm in two phases-design and implementation. In design phase, we analyze the algorithm, determine the bottleneck, re-factor the algorithm and data to enhance parallelism. For implementation phase, we follow the standard guidelines for optimization on GPU prescribed by NVIDIA and GPU experts, and fine tune GPU architecture parameters such as threading, blocks, streams etc. to obtain the near optimal implementation. Our method of parallelizing the algorithm first via design, and then via implementation, makes our effort of parallelization different from other

works.

The original work presented in [1] was written by one of our authors. The work has been used in [2] and also has been acknowledged as a highly accurate pose estimation algorithm in [3] [4]. However, no effort has been made to parallelize the algorithm described in [1]. There are several real-time suitable pose estimation algorithms published in the year 2015 alone, [5]-[11] to mention a few. However, these algorithms have limitations based on their approach. Pose estimation described in [5] [6] [7] [8] require a database of objects. The limitation of using database centered pose estimation is the overhead involved in creating a useful, open source and widely accepted database. Moreover, the pose estimation application would be limited to the objects in the database. The pose estimation algorithm described in [8] [9] exploit certain features for a given object. The usage of such algorithm is limited to certain objects, thereby limiting its application. The pose estimation described in [10] requires special cameras. The cameras are an additional cost, and may not be feasible to use in different kinds of application. On the contrary, the algorithm described in [1] is generic pose estimation algorithm with high accuracy, *i.e.*, it does not require a database or exploit certain features of an object or use special cameras. In comparison to other generic and highly accurate pose estimation algorithms, it estimates pose without subjecting the estimation to local minima or divergence [1].

The focus of our research work is to parallelize the algorithm in [1]. While the sequential implementation of the algorithm in [1] can serve as a baseline for measuring our parallelization, we have reviewed four GPU implemented pose estimations, generic and non-generic, to compare our parallelization and provide a reference to readers. These reviewed implementations have low execution time or high accuracy or both as their landmark. The proposed implementation in [11] accelerates a generic pose estimation algorithm on GPUs, completing detection and pose estimation under 22 ms per image of resolution 500 × 500 pixels. The estimated pose is not as accurate as the one described in [1], but is comparable to our implementation. The algorithm in [12] provides another generic pose estimation algorithm implemented on GPUs. The pose estimation is reconstructed in real time, with execution time varying from 10 milliseconds to hundreds of milliseconds depending on the number of objects, the kind of object, the resolution and the accuracy. This parallel implementation is comparable to our work, with a difference that our work provides more accurate pose estimation. The pose estimation algorithm described in [13], is highly accurate human pose estimation using datasets. The algorithm in [13] is implemented on GPUs for low resolution images, with execution time varying between 100 milliseconds to 1200 milliseconds, based on the accuracy of pose estimation. Hence, the algorithm in [13] is limited to human pose estimation, applicable for low-resolution images and has longer execution time than our implementation. The algorithm described in [14] is used for highly accurate pose estimation in un-

manned air vehicles (UAV) for takeoff and landings. The authors in [14] perform a field evaluation using low end GPUs such as Jetson TK1. It is observed that pose estimation takes under 25 milliseconds for standard 640 × 480 images. Though the algorithm has good performance, especially considering that the simulations were carried out on a low-end GPU, it is explicitly used for UAV.

The purpose of our work is to extend the work in [1]. Parallelizing the algorithm in [1] would help us build a highly accurate, robust, real-time feasible and scalable algorithm, which would make our parallel version of pose estimation algorithm unique, with respect to all other algorithms that have only few, but not all, of the above mentioned features.

This paper is organized as follows: Section 2 describes an implementation of the algorithm provided in [1]. Section 3 provides the optimization design. In Section 4, the performance of our parallel implementation is compared with sequential execution and a parallel implementation of the continuous 8-point pose estimation algorithm. We conclude the paper by summarizing our findings.

2. Pose Estimation Algorithm

The implementation description of the pose estimation algorithm introduced in [1] henceforth known as the optimal position estimation algorithm (OPEA) is provided in this section. The following assumptions for the implementation are made:

a) Monocular pose estimation is considered.

b) There is a relative motion between the camera and object.

c) There are at least four markers per object.

d) Camera captures images in the form of video frames, with at least 24 frames per second (fps). The fps is assumed to increase with an increase in the rate of relative motion between the camera and the object.

e) It is a continuous pose (ω, v) estimation, where, $\omega = \begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}^T$ is the angular velocity relative to the camera and object, $v = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}^T$ is the translational velocity relative to the camera and the object

f) The velocity, v, is measured in spherical coordinates. Spherical coordinates (r, β, φ) are represented by radial distance (r), azimuthal angle (β) and polar angle (φ) as shown in **Figure 1**. By assuming $-\frac{\pi}{2} \le \beta \le \frac{\pi}{2}$, $-\frac{\pi}{2} \le \varphi \le \frac{\pi}{2}$ and $r = 1$, an iterative search can be performed with ease and then scaled by a factor of radial distance.

Since OPEA is an iterative 2D search algorithm, it explores a 2D space for all possible values of the pose of v and ω. The search begins by assuming a value of v, where v in spherical coordinates is given by $v = \begin{bmatrix} \sin\varphi\cos\beta & \sin\varphi\sin\beta & \cos\varphi \end{bmatrix}$, and $r = 1, -\frac{\pi}{2} \le \beta \le \frac{\pi}{2}, -\frac{\pi}{2} \le \varphi \le \frac{\pi}{2}$. The magnitude of v is inherently ambiguous in the image data. It is impossible to find r. Consequently, it is set to be r

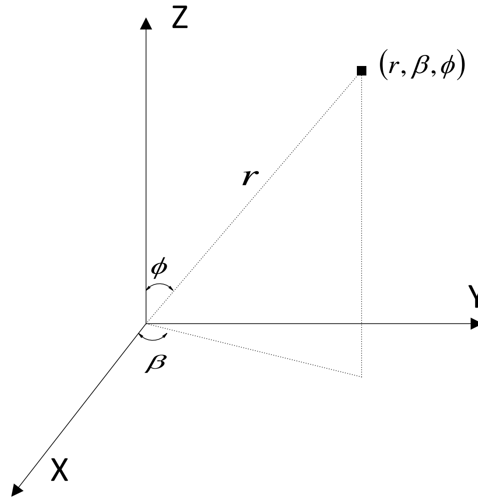

Figure 1. Spherical coordinate system representation.

= 1. For each assumption of *v*, a corresponding ω is calculated. The error in (ω, v) pair estimation is calculated by re-projecting all the markers of the object. The error is quantized and compared across all values of (ω, v). The (ω, v) providing the least error and positive depths for all markers is selected as the pose. Since the value determining v have boundary conditions $-\frac{\pi}{2} \le \beta \le \frac{\pi}{2}$, $-\frac{\pi}{2} \le \varphi \le \frac{\pi}{2}$, the increment intervals of β and φ determine the accuracy of the pose estimated, *i.e.*, the smaller the incremental value of β and φ, the greater the accuracy of (ω, v).

Consider two images taken from a single camera with relative velocity between the image and the object. Let the image coordinates of markers on the first image be x_l, where $x = [x \quad y \quad 1]^{\mathrm{T}}$ and $l = 4, \cdots, m_f$ and m_f represents the number of markers per image frame. Let the difference between the markers' positions on the second frame from the first frame be represented by $\dot{x}_l$, where $\dot{x} = [\dot{x}_1 \quad \dot{x}_2 \quad 0]^{\mathrm{T}}$. Unit vectors for x_l and $\dot{x}_l$ are calculated using:

$$u_l = \frac{x_l}{\|x_l\|} \tag{1}$$

$$\dot{u}_l = \left(I - u_l u_l^{\mathrm{T}}\right) \frac{\dot{x}_l}{\|x_l\|} \tag{2}$$

where I represents an identity matrix and u_l^{T} represents the transpose of u_l.

Once a value of v is assumed, a corresponding value for ω is calculated as follows:

$$\omega = -C^{-1} B \tag{3}$$

$$C = \sum_{l=1}^{m_j} \hat{u}_l^2 v v^{\mathrm{T}} \hat{u}_l^2 \tag{4}$$

$$B = \sum_{l=1}^{m_j} \hat{u}_l^2 v v^{\mathrm{T}} \hat{u}_l \dot{u}_l \tag{5}$$

$$\hat{u} = \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix} \tag{6}$$

$$u = \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} \tag{7}$$

The difference $\left(\dot{u}^i - \hat{\omega} u^i\right)$ provides the error measurement in ω. We obtain the cross product of this difference with u to obtain a perpendicular vector, representative of the error in estimating pose. This process removes the depth of all points from the equations. A single scalar value error (J) for all markers of an object, indicating the error in estimating (ω, v) is calculated as follows:

$$Q_s = \sum_{l=1}^{m_j} \left[\hat{u}_l \left(\dot{u}_l - \hat{\omega} u_l\right)\right] \left[\hat{u}_l \left(\dot{u}_l - \hat{\omega} u_l\right)\right]^{\mathrm{T}} \tag{8}$$

$$J = v^{\mathrm{T}} Q_s v \tag{9}$$

Thus, the plot of J versus (β, φ), selected for v, provides a surface. The lowest points (β, φ) with respect to J on the surface show possible (ω, v) values with lowest errors. Similar to other pose estimations, depth constraints dictated by the epipolar geometry helps in selecting the right pose. The depth (d) constraint for each marker is calculated as follows:

$$d_l = \left(\dot{u}_l - \hat{\omega} u_l\right)^{\mathrm{T}} v \tag{10}$$

If the depth of all points has the positive numerical sign and the value of J corresponding to the same (ω, v) is minimal, then (ω, v) is taken as the pose. If the depth of all points has the negative numerical sign and the value of J corresponding to the same (ω, v) is minimal, then the pose is taken to be $(\omega, -v)$.

3. Parallelization of OPEA

We begin this section by analyzing the calculations in OPEA. The analysis is split into two phases. The first phase analyzes code refactoring of OPEA to suit parallelization. However, the refactored code can be used as a sequential or parallel code. The second phase analyzes the data reorganization and code for a parallel implementation of the refactored code, to match the GPU architecture.

The first phase of analysis begins by splitting the calculations in OPEA into three steps. Equations (1)-(5) are taken to be the "compute velocity" step. Equations (8), (9) are taken to be "compute error" step. Lastly, Equation (10) is taken to be the "selection of pose" step. If the total number of iterations in each dimension is represented by k, Table 1 shows the number of floating point of operations (FLOP) for each step.

Hence the execution time of OPEA increases linearly with an increase in m_f but a quadratic increase with an increase in k.

Table 1 also shows that the compute velocity step has the maximum FLOP count in the computation phase. Analyzing Equations (1)-(7), the dependency of variables for the compute velocity step is provided in Figure 2. Clearly, the compute velocity step in OPEA has varied data dependencies, which can be separated as:

a) v dependent computations

b) v independent computations

Since only v dependent computations are needed inside the iteration, the v independent computations can be pre-computed outside the iterations, thus reducing the total FLOP count of OPEA.

We now provide an advanced version of OPEA, referred to as AOPEA, which refactors code to suit parallelization better. To facilitate code refactorization, we introduce two new operations

a) Stackoperation: Convert 3 × 3 symmetric matrix elements to a vector.

b) Unstackoperation: Convert 6 × 1 vector into 3 × 3 symmetric matrix.

Table 1. FLOP Count Split-up for OPEA Algorithm.

Step	FLOP count
Compute Velocity	$138m_f + 49$
Compute Error	$45m_f + 11$
Selection of Pose	$5m_f$
Per Iteration	$188m_f + 60$
OPEA Algorithm	$(188m_f + 60)k^2$

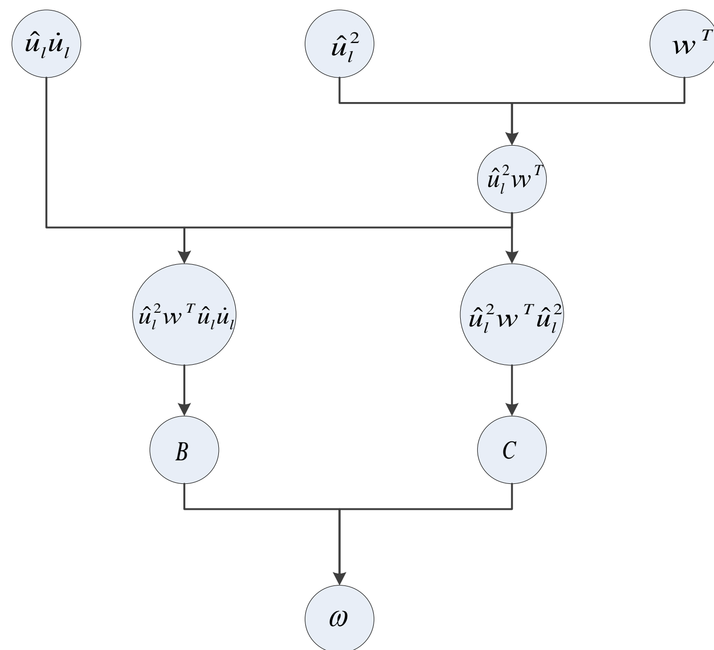

Figure 2. Dependencies in compute velocity step.

$$\begin{bmatrix} s_1 & s_2 & s_3 \\ s_2 & s_4 & s_5 \\ s_3 & s_5 & s_6 \end{bmatrix} \underset{unstack}{\overset{stack}{\rightleftarrows}} \begin{bmatrix} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \end{bmatrix}$$

The computation of B, C and Q_s are performed in two parts. The v independent computations, computed only once for a given set of markers, represented by B_p, C_p^s, Q_{sp}, and intermediary result variable M_p are calculated as shown below:

$$B_p = \sum_{l=1}^{m_f} \begin{bmatrix} c_1^l \mu_1^l \\ c_1^l \mu_2^l + c_2^l \mu_1^l \\ c_1^l \mu_3^l + c_3^l \mu_1^l \\ c_2^l \mu_2^l \\ c_2^l \mu_3^l + c_3^l \mu_2^l \\ c_3^l \mu_3^l \end{bmatrix}^{\mathrm{T}} \tag{11}$$

$$C_p^s = \begin{bmatrix} \sum_{l=1}^{m_f} \overline{u}_1^l \\ \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \sum_{l=1}^{m_f} \overline{u}_2^l \\ \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \sum_{l=1}^{m_f} \overline{u}_3^l \end{bmatrix} \tag{12}$$

$$Q_{sp}^j = \sum_{l=1}^{m_f} \left(\hat{u}_l^2 \left(\hat{u}_l \dot{u}_l \right) \right)_j \tag{13}$$

$$M_p = \sum_{l=1}^{m_f} \left(\hat{u}_l \dot{u}_l \right) \left(\hat{u}_l \dot{u}_l \right)^{\mathrm{T}} \tag{14}$$

where,

$$c^l = \left(\hat{u}_l \dot{u}_l \right) \tag{15}$$

$$\hat{u}_l^2 = \begin{bmatrix} \mu_1^l & \mu_2^l & \mu_3^l \end{bmatrix} \tag{16}$$

$$\overline{u}_j^l = \begin{bmatrix} \mu_{j1}^l \mu_1^l \\ \mu_{j1}^l \mu_2^l + \mu_{j2}^l \mu_1^l \\ \mu_{j1}^l \mu_3^l + \mu_{j3}^l \mu_1^l \\ \mu_{j2}^l \mu_2^l \\ \mu_{j2}^l \mu_3^l + \mu_{j3}^l \mu_2^l \\ \mu_{j3}^l \mu_3^l \end{bmatrix}^{\mathrm{T}} \tag{17}$$

where c_j^l represents the jth element of the vector c^l and μ_1, μ_2, μ_3 are vectors. The v dependent computations, which are performed for all iterations, to complete the calculations of B, C and Q_s are provided below:

$$B = B_p v^S \tag{18}$$

$$C = \text{unstack}\left(C_p^s v^S \right) \tag{19}$$

$$Q_s = M_p + N + N^{\mathrm{T}} + C \tag{20}$$

where,

$$v^S = \text{stack}\left(vv^{\mathrm{T}}\right) \tag{21}$$

$$N = \begin{bmatrix} Q_{s1}\omega & Q_{s2}\omega & Q_{s3}\omega \end{bmatrix} \tag{22}$$

The new FLOP count for the AOPEA is provided in **Table 2**. In comparison with OPEA, the number of computations is observed to be significantly lower.

The steps for selection of pose, discussed in Equation (10), can be skipped if J is greater than an already computed minimum value. Hence, neglecting the FLOP count for the same, the total FLOP count for AOPEA is $177m_f + 216k^2 - 24$. The separation of m_f and k^2 computational steps facilitates an increase in execution time exclusive to either the number of markers or accuracy. For example, a real-time application requiring higher accuracy can have lower m_f and high k value, whereas a real-time application requiring higher m_f can have low k iteration count. In both cases, only the computations relevant to each are increased.

To reduce the overhead of iterations further; the v dependent computations are performed twice-first with coarse accuracy with high increments for each iteration, and second with finer accuracy with smaller increments for each iteration. Coarse accuracy iterations, having smaller values of k, provide a coarse estimation of (ω, v) which helps in limiting the search to a smaller interval. The second round of v dependent computations are performed with finer accuracy values for a much smaller interval around the coarse estimated (ω, v). These modifications result in refactoring the OPEA code to AOPEA algorithm along with refactored code. Thus, this reduces the total number of iterations performed significantly. The increment values for the coarser and finer iterations are dependent on the application. The AOPEA can be implemented either as a sequential or parallel algorithm.

The second phase of analysis looks at the implementation of AOPEA as a parallel algorithm. It begins with looking at data reusability for the algorithm. **Figure 3** shows how some of the computed data variables can repeatedly be used.

Table 2. FLOP count Split-up for AOPEA algorithm.

Step	v Independent FLOP count	v Dependent FLOP count
Compute Velocity	$141m_f - 12$	160
Compute Error	$36m_f - 12$	56
Selection of Pose	0	$5\ m_f$
Per Iteration	0	$5\ m_f + 216$
AOPEA Algorithm	$177m_f - 24 + (5m_f + 216)k^2$	

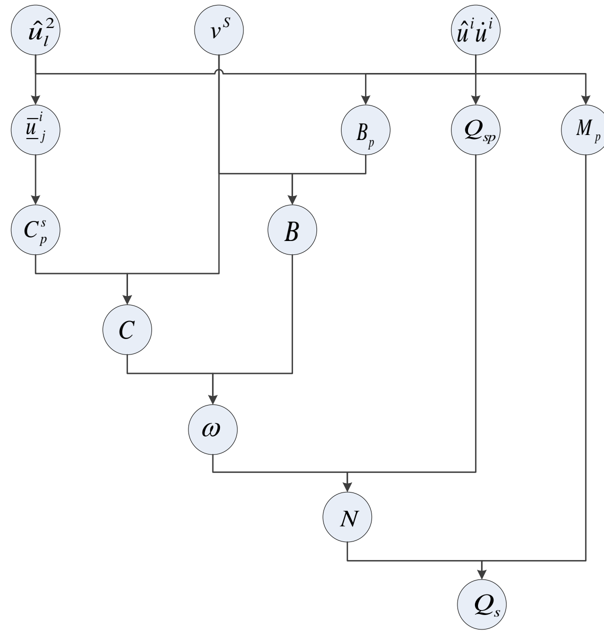

Figure 3. AOPEA data reusability.

For the v independent computations, we arrange the data into two matrices such that one matrix-matrix multiplication provides the results for B_P, Q_{sp} and m_p. We perform a global synchronization after v independent computations are completed. The results thus obtained are re-arranged into a single matrix, and distributed (data) to many-cores on GPUs, known as streaming processors. This arrangement facilitates coalesced memory access for all matrix or vector multiplications and additions involved. Coalesced memory access on GPUs, are shown to provide better performance in [15]. Since iterations are mutually exclusive, we can assign one iteration search to a single streaming processor on the GPU. Apart from separating the iterations, GPU-based code optimizations as shown in [15] are performed on the code for maximum performance.

4. Results

In this section, we analyze the effectiveness of AOPEA, its parallelization, and scalability. To analyze AOPEA, first, we compare the execution time of sequential implementation of AOPEA with a reference pose estimation algorithm, and sequential OPEA algorithm. For the reference pose estimation, we use the continuous pose estimation algorithm (CPEA). Second, we look at the execution time of parallel implementations of CPEA, OPEA, and AOPEA parallelized using standard parallel libraries. Lastly, we look at the execution time of our parallel implementation of AOPEA, analyzing its scalability.

The programs are executed on Intel Xeon CPU, having two E5620 processors operating at 2.40 GHz and running a 64 bit Windows 7 Pro Operating System. The sequential programs are executed on a 64-bit MatlabR2014b software.

Matlab'soriginal core has been developed from LINPACK and EISPACK [16]. LINPACK and EISPACK have proven to be computationally effective ways to solve linear algebra problems [17]. Hence we use this software to obtain the reference time for sequential programs.

For parallel program execution, the CPU is equipped with a NVIDIA Tesla C2075 card. The card is equipped with 448 cores and 6GB of memory for general computations. Though K20x and K40m cards seem to be a good option for GPU parallelization, we limit ourselves to a low-cost GPU such as C2075. For the development of parallel AOPEA implementation, NVIDIA's CUDA 6.0 integrated with Microsoft Visual 2015 via NSight was used. For developing parallel CPEA, OPEA and AOPEA code using standard libraries, cuBLAS library package was used. cuBLAS library package is an accelerated Basic Linear Algebra Subprograms (BLAS) library provided by NVIDIA for GPUs. A combination of APIs available in the cuBLAS package is used to perform the operations in CPEA, OPEA, and AOPEA. Lastly, for profiling NVIDIA's Visual Profiler, NVVP, compatible with CUDA 6.0 and Microsoft Visual 2015 was used to profile the code.

For the purpose of this paper, we use execution time as a measure of performance. Low execution time is considered to be better. Each implementation of an algorithm (CPEA, OPEA or AOPEA) for a given marker size and accuracy is considered as one simulation and the execution time is collected. Each simulation is executed one thousand times and an average execution time (AET)is computed. The standard deviation of AET for all simulations was observed to be under 3%. Each simulation has been verified by reconstructing objects in images. Data, for verification of algorithms, is taken from videos under indoor computer lab conditions using 60 fps at 1080 p resolution. The number of markers for simulations is varied to study the scalability of the algorithms, *i.e.*, we choose $m_f = \left[2^5, 2^6, \cdots, 2^{11}\right]$. For all simulations, the accuracy is assumed to be 0.01 radians which satisfies the accuracy required for 3D re-projection for real-time applications. For AOPEA, the accuracy for the coarse resolution is taken to be 0.15 *radians*, and 0.01 *radians* for finer resolution. The interval for the finer search is taken to be twice the coarse resolution.

Table 3 shows the sequential AET of CPEA, OPEA and AOPEA algorithms for a different number of markers per frame. For smaller m_f, the CPEA has lower AET, whereas, for larger m_f, the CPEA has a non-linear increase in AET. This is due to the computations in CPEA being $O\left(\left(m_f\right)^3\right)$. Hence for small m_f, AET for CPEA is low and grows exponentially as m_f increases. The computations of OPEA, as seen in Table 1, are $O\left(m_f k^2\right)$. Hence, the AET for OPEA is significantly higher for all m_f values. For AOPEA, the AET is higher than CPEA for lower values of m_f values. This is due to the overhead in v independent computations that are pre-computed before the iterations. However, as v increases, there is a proportionate increase in AET. This is due to the iterative computations of AOPEA being linearly proportional to m_f, as seen in Table 2.

Next, we look at the parallel implementation of CPEA, OPEA and AOPEA, implemented using cuBLAS library API calls. The results, which include the data transfer time between CPU and GPU, are presented in Figure 4. The CPEA, after parallelization, shows reduced AET in comparison to sequential AET, showing

Table 3. Average execution time for sequential CPEA, OPEA, and AOPEA algorithms for different markers per frame.

Markers per Frame	AET (msecs)		
	CPEA	OPEA	AOPEA
32	0.32	1.52E+05	97.70
64	1.10	2.97E+05	102.50
128	1.80	5.84E+05	118.08
256	8.50	1.18E+06	146.14
512	41.60	2.31E+06	194.74
1024	824.50	4.66E+06	296.45
2048	70826.70	1.02E+07	503.71

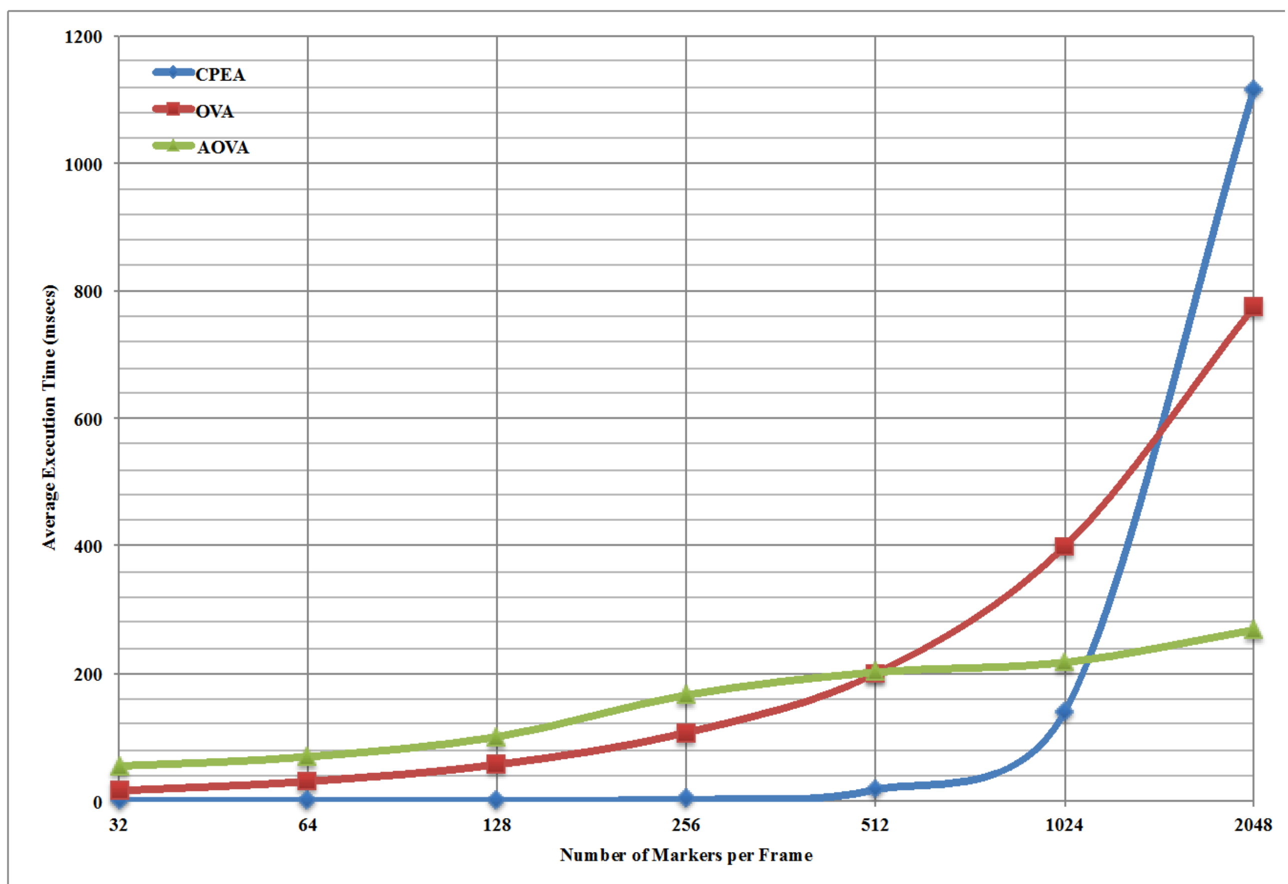

Figure 4. Average execution time comparison between parallel implementation of CPEA, OPEA and AOPEA using cuBLAS for Different Number of Markers.

good scaling till 512 m_f. This is because CPEA has sufficient parallelism in its code. However, for $m_f > 512$ the profiler indicated occupation of all cores on the GPU. This forces fragments of code in the queue to wait till cores become idle serializing the execution, increasing the AET. In the case of OPEA, the algorithm is embarrassingly parallel with respect to iterations *i.e.*, the number of iterations is more than 90,000 and independent of one another. Hence, the problem has enough workload for GPUs even for low m_f. The profilers indicate that the GPU cores are completely occupied. But, as m_f increases, due to the lack of idle GPU cores, more code execution is serialized leading to higher AET. In case of AOPEA, with lower computations than CPEA and OPEA and increasing linearly with increase in m_f, the AET shows small increments for every doubling of m_f. Due to the separation of v dependent and independent computations in AOPEA, the data needs to be re-arranged after completing the v independent computations. This forces additional data transfers between CPU and GPU. For low values of m_f, the AET is limited by the overhead of data transfers between CPU and GPU. In fact, the data transfers contribute to nearly 90% of AET for all values of m_f. However, despite overhead of data transfer time, AOPEA has the best AET for higher values of m_f.

Figure 5 shows the AET of our version of AOPEA for different m_f values. Due

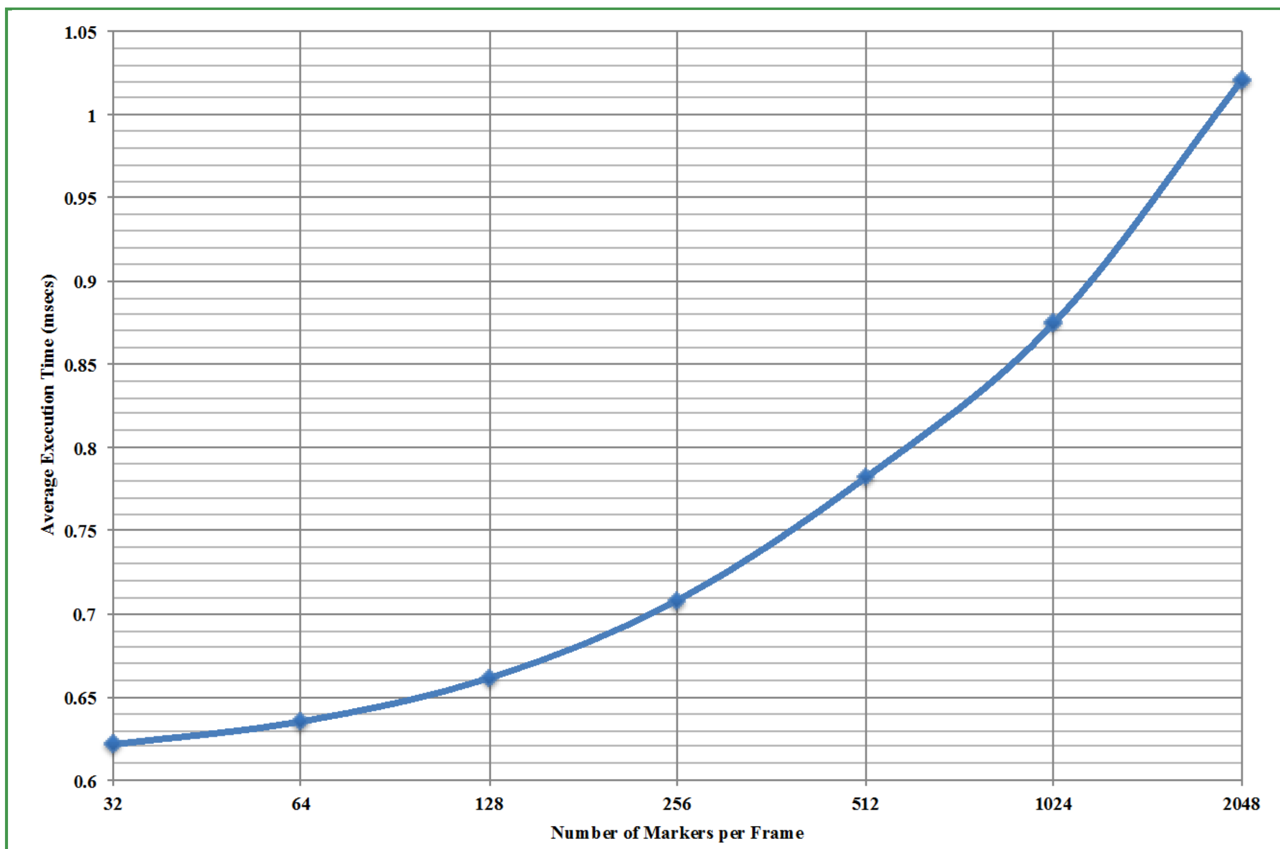

Figure 5. Average execution time of our implementation of AOPEA different number of markers.

to a single data transfer between CPU and GPU in each direction, and highly optimized code, the AET is found to be just over a millisecond for 2048 markers. The implementation also shows good scalability even at 2048 markers, unlike Figure 4 *i.e.*, with an increase in markers there is a linear increase of AET. To better compare the performance of our version of parallel AOPEA, Figure 6 provides the speed up of our parallel implementation of AOPEA with CPEA, OPEA and AOPEA implemented using cuBLAS library calls. In case of CPEA, the speed up exponentially increases, especially for higher m_f. In case of OPEA, we observe a linear speed up with an increase in markers. Whereas for AOPEA, the speed up saturates at about 250×. The low AET and good scalability of our parallel implementation of AOPEA indicate its suit ability for real-time applications.

Though our version of pose estimation shows low AET, using it for real-time applications may have higher AET. This is because real-time applications combine pose estimation with tracking of markers. This would involve additional overheads. For example, real-time applications would use a video, where each frame is considered as an image. The image from the camera needs to be transferred to CPU, and then to the GPU. The image may also need to be preprocessed to obtain distinct position of markers. In order to show that our parallel

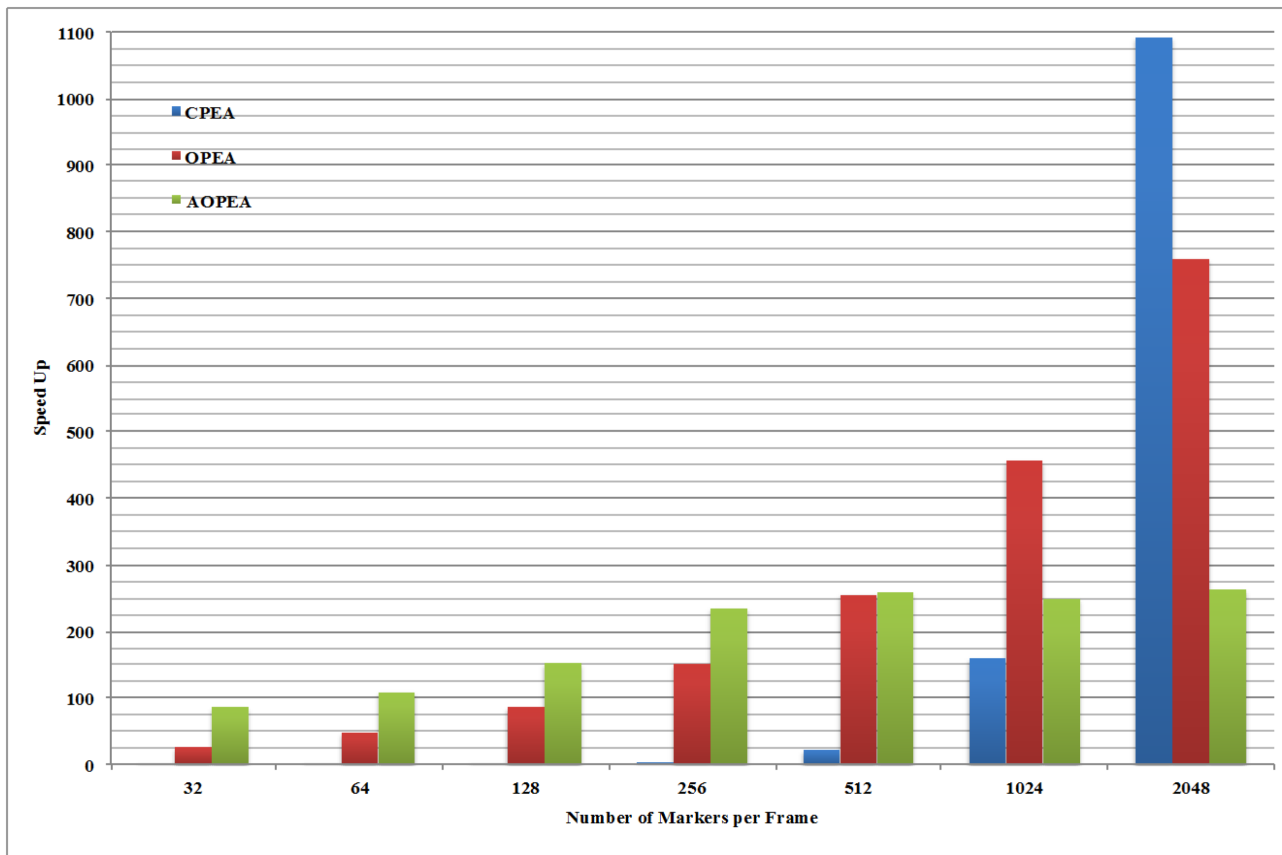

Figure 6. Speed up of our implementation in comparison to CPEA, OPEA & AOPEA implemented using cuBLAS library calls.

implementation can be used for real time applications, we conducted a lab experiment. A 1000 fps high-resolution camera was used to capture a video tracking 32 markers. The position of the 32 markers were pre-calculated, but fed to the pose estimation algorithm in real time on a per image basis. Simulations for different image resolutions were conducted, where each image obtained from the camera, was copied to CPU, transferred to GPU, converted from Red-Green-Blue format to gray-scale format, and then pose was estimated. Using simulations' AET, the supported fps that our pose estimation algorithm could process, was calculated.

Figure 7 shows the supported fps by our parallel implementation of AOPEA for 640 × 480 (640 k), 1280 × 720 (720 p) and 1920 × 1080 (1080 p) resolution videos. For 640 k videos, real-time applications using 600 fps can use our parallel implementation with ease since our implementation supports 681 fps. However, for 720 p and 1080 p, due to the large amount of data, we observe a sharp decrease in supported fps. For 720 p, our parallel implementation of AOPEA supports 285 fps, whereas for 1080 p, it supports 112 fps.

5. Conclusions

We have modified the algorithm provided in [1] to meet the performance

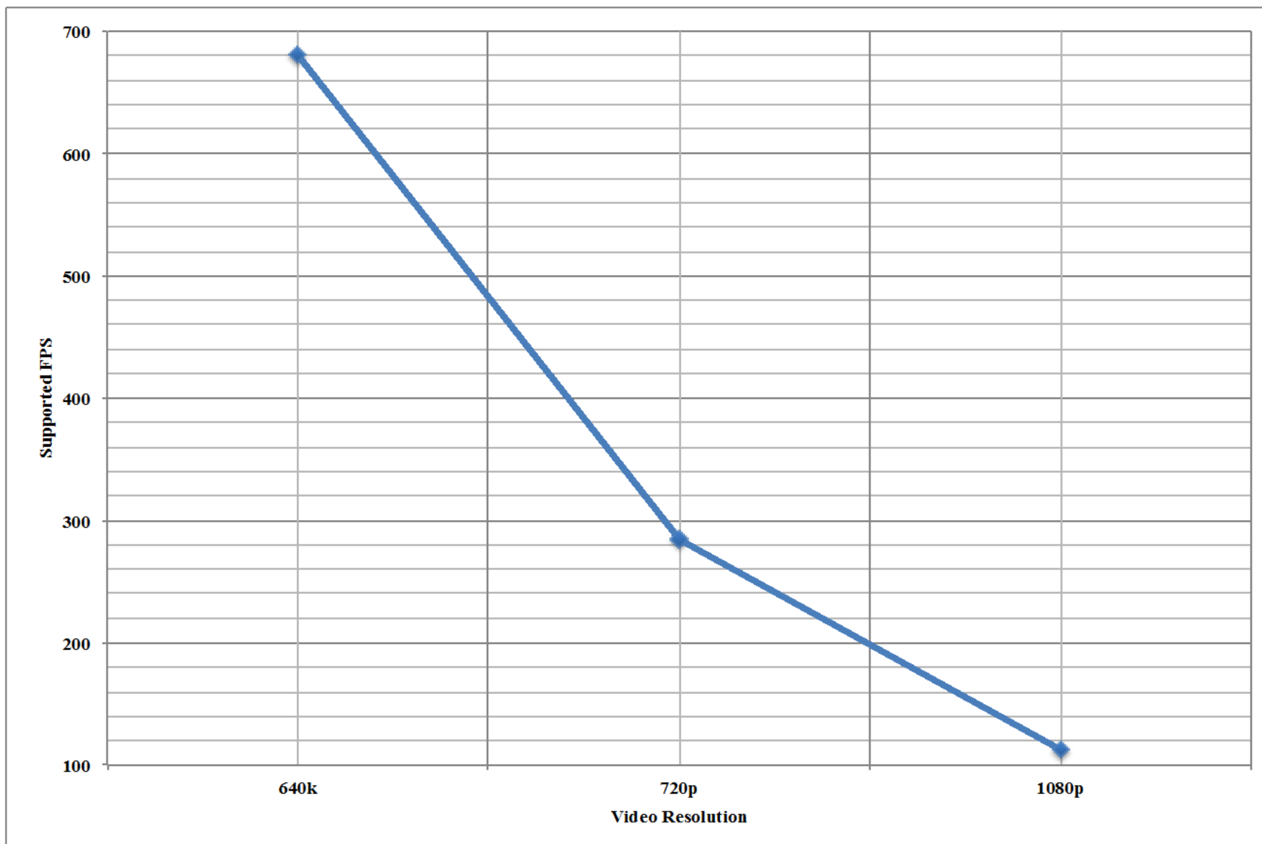

Figure 7. Supported FPS for real-time pose estimation using our parallel implementation of AOPEA.

requirements of real-time applications. An analysis of the implementation indicated that there was a) redundancy in computations and b) data and code organization un-fitting for GPU architectures. We modified the implementation, to suit parallelization on GPU architectures, in two phases: first, refactoring the algorithm to have lesser number of operations and enhanced parallelism, and secondly, optimizing the data and code to obtain better parallelism for GPU architectures. We compared the effectiveness of our algorithm AOPEA, with CPEA and OPEA, for sequential and parallel implementations. For sequential implementation, AOPEA performed much better than its predecessor algorithm OPEA. However, for lower markers per frame, CPEA performed better than AOPEA whereas for higher markers per frame AOPEA performed better than CPEA. To understand the effectiveness of our parallel implementations, CPEA, OPEA and AOPEA were parallelized using the cuBLAS library and compared with our parallel implementation. The results showed that our parallel implementation of AOPEA has lowest execution time. Moreover, our parallel implementation also showed good scalability of performance with an increasing number of markers per frame. Moreover, a lab simulation of a real-time application indicated that our parallel implementation of AOPEA supports at least 100 fps even for high-resolution videos. Hence, our parallel implementation of AOPEA could be implemented on GPUs for real-time applications using high-resolution frames with a high number of markers per frame.

For our future work, we plan on pursuing two additions to the AOPEA: a) multi-GPU implementation of the algorithm for images from 3 - 4 cameras, and b) integrating our AOPEA and multi-GPU AOPEA algorithm with a tracking algorithm.

References

[1] McInroy, J.E. and Qi, Z. (2008)A Novel Pose Estimation Algorithm Based on Points to Regions Correspondence. *Journal of Mathematical Imaging and Vision*, **30**, 195-207. https://doi.org/10.1007/s10851-007-0045-2

[2] Qiao, B., Tang, S., Ma, K. and Liu, Z. (2013) Relative Position and Attitude Estimation of Spacecrafts Based on Dual Quaternion for Rendezvous and Docking. *Acta Astronautica*, **91**, 237-244. https://doi.org/10.1016/j.actaastro.2013.06.022

[3] Yang, D., Liu, Z., Sun, F., Zhang, J., Liu, H. and Wang, S. (2014) Recursive Depth Parametrization of Monocular Visual Navigation: Observability Analysis and Performance Evaluation. *Information Sciences*, **287**, 38-49. https://doi.org/10.1016/j.ins.2014.07.025

[4] Yang, D., Sun, F., Wang, S. and Zhang, J. (2014) Simultaneous Estimation of Ego-Motion and Vehicle Distance by Using a Monocular Camera. *Science China Information Sciences*, **57**, 1-10. https://doi.org/10.1007/s11432-013-4884-8

[5] Whelan, T., Kaess, M., Johannsson, H., Fallon, M., Leonard, J.J. and McDonald, J. (2015) Real-Time Large-Scale Dense RGB-D SLAM with Volumetric Fusion. *The International Journal of Robotics Research*, **34**, 598-626. https://doi.org/10.1177/0278364914551008

[6] Gálvez-López, D., Salas, M., Tardós, J.D. and Montiel, J.M.M. (2016) Real-Time Monocular Object Slam. *Robotics and Autonomous Systems*, **75**, 435-449. https://doi.org/10.1016/j.robot.2015.08.009

[7] Lim, H., Sinha, S.N., Cohen, M.F., Uyttendaele, M. and Kim, H.J. (2015) Real-Time Monocular Image-Based 6-DoF Localization. *The International Journal of Robotics Research*, **34**, 476-492. https://doi.org/10.1177/0278364914561101

[8] Tagliasacchi, A., Schröder, M., Tkach, A., Bouaziz, S., Botsch, M. and Pauly, M. (2015) Robust Articulated-ICP for Real-Time Hand Tracking. *Computer Graphics Forum*, **34**, 101-114. https://doi.org/10.1111/cgf.12700

[9] Rymut, B. and Kwolek, B. (2015) Real-Time Multiview Human Pose Tracking Using Graphics Processing Unit-Accelerated Particle Swarm Optimization. *Concurrency and Computation*: *Practice and Experience*, **27**, 1551-1563. https://doi.org/10.1002/cpe.3329

[10] Sun, K., Heß, R., Xu, Z. and Schilling, K. (2015) Real-Time Robust Six Degrees of Freedom Object Pose Estimation with a Time-of-Flight Camera and a Color Camera. *Journal of Field Robotics*, **32**, 61-84. https://doi.org/10.1002/rob.21519

[11] Poirson, P., Ammirato, P., Fu, C.Y., Liu, W., Kosecka, J. and Berg, A.C. (2016) Fast Single Shot Detection and Pose Estimation.

[12] Zabulis, X., Lourakis, M.I. and Koutlemanis, P. (2016) Correspondence-Free Pose Estimation for 3D Objects from Noisy Depth Data. *The Visual Computer*, 1-19. https://doi.org/10.1007/s00371-016-1326-9

[13] Haque, A., Peng, B., Luo, Z., Alahi, A., Yeung, S. and Fei-Fei, L. (2016) Towards Viewpoint Invariant 3D Human Pose Estimation. 14*th European Conference on Computer Vision*, Amsterdam, 11-14 October 2016, 160-177. https://doi.org/10.1007/978-3-319-46448-0_10

[14] Benini, A., Rutherford, M.J. and Valavanis, K.P. (2016) Real-Time, GPU-Based Pose Estimation of a UAV for Autonomous Takeoff and Landing. *IEEE International Conference on Robotics and Automation*, Stockholm, 16-21 May 2016, 3463-3470. https://doi.org/10.1109/icra.2016.7487525

[15] Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B. and Hwu, W.M.W. (2008) Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. *Proceedings of the* 13*th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, 20-23 February 2008, 73-82. https://doi.org/10.1145/1345206.1345220

[16] Ramaswamy, S., Hodges, E.W. and Banerjee, P. (1996) Compiling Matlab Programs to ScaLAPACK: Exploiting Task and Data Parallelism. *Proceedings of the* 10*th International Parallel Processing Symposium*, Honolulu, 15-19 April 1996, 613-619. https://doi.org/10.1109/ipps.1996.508120

[17] Dongarra, J.J., Bunch, J.R., Moler, C.B. and Stewart, G.W. (1979) LINPACK Users' Guide. SIAM. https://doi.org/10.1137/1.9781611971811

Submit or recommend next manuscript to SCIRP and we will provide best service for you:

Accepting pre-submission inquiries through Email, Facebook, LinkedIn, Twitter, etc.
A wide selection of journals (inclusive of 9 subjects, more than 200 journals)
Providing 24-hour high-quality service
User-friendly online submission system
Fair and swift peer-review system
Efficient typesetting and proofreading procedure
Display of the result of downloads and visits, as well as the number of cited articles
Maximum dissemination of your research work

Submit your manuscript at: http://papersubmission.scirp.org/
Or contact jcc@scirp.org