Implementation of the Lucas-Kanade Image Registration Algorithm on a GPU for 3D Computational Platform Stabilisation

Bernardt Duvenhage[*], JP Delport[†] and Jason de Villiers[‡]
Council for Scientific and Industrial Research

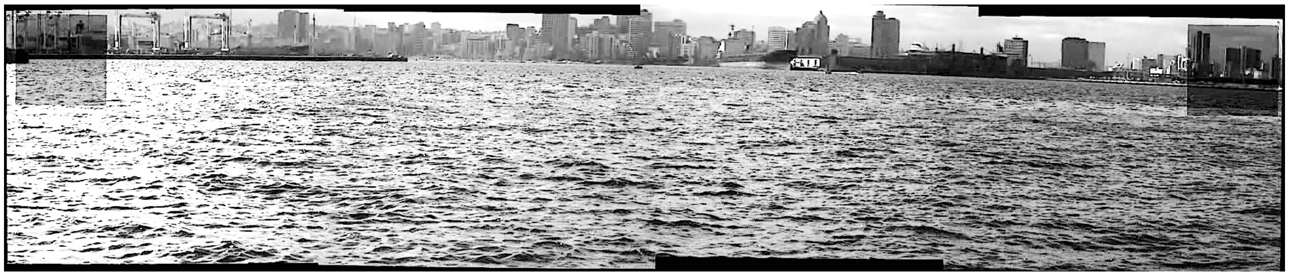


Figure 1: *Wide-Angle View of the Durban Coastline - Stitched and Stabilised*

Abstract

Image registration forms the basis of many computer vision tasks. The Lucas-Kanade image registration algorithm is known to efficiently solve the sub-problem of *rigid* image registration. It is therefore often used in image stabilisation applications. This paper presents the details of a real-time implementation of the Lucas-Kanade image registration algorithm on a Graphics Processing Unit (GPU) using the OpenGL Shading Language (GLSL). The implementation is driven by a real world requirement to computationally stabilise the undulatory motion of an ocean-based wide area surveillance system.

CR Categories: I.4.3 [Computing Methodologies]: IMAGE PROCESSING AND COMPUTER VISION—Enhancement;

Keywords: Lucas-Kanade, Image Stabilisation, Graphics Processing Unit

1 Introduction

This paper investigates an implementation of the Lucas-Kanade image registration algorithm on a parallel Graphics Processing Unit (GPU). The implementation requirement stems from the need to computationally stabilise the undulatory motion of a real-time ocean-based wide area surveillance system. The background to the problem is introduced followed by the scope of the work and the structure of the paper.

[*]e-mail: bduvenhage@csir.co.za
[†]e-mail: jpdelport@csir.co.za
[‡]e-mail: jdvilliers@csir.co.za

1.1 Background

The need to do real-time image registration arose from a real-world requirement to do 3D computational platform stabilisation in a Wide Area Surveillance Prototype (WASP). WASP is aimed at providing an automated all-weather, 24 hour, omnidirectional solution to fulfil the surveillance needs of the South African (SA) Armed Forces, with the SA Navy being the primary client. An array of cameras staring radially outward provides up to a full 360° surveillance capability. The system is capable of detecting and tracking targets that are traditionally difficult for Radio Detection And Ranging (RADAR) sensors to detect (such as small wooden craft only a few hundred meters away). By using the latest low-light cameras, or increasingly inexpensive infrared cameras, night and adverse weather conditions can also be catered for. High resolution video streams from synchronised cameras are seamlessly stitched into a single wide-angle image. This is then followed by foreground/background separation, target detection, target tracking and motion prediction, and threat assessment as appropriate. Image registration is used to compute the transform that would warp the wide-angle image to remove the undulatory motion of the platform. Removing the effects of the platform motion in this way computationally stabilises the surveillance system for effective foreground/background separation and tracking.

The release of version 1.2 of the Open Graphics Library (OpenGL) in March of 1998 provided an optional imaging subset of commands to enable parallel convolutions on images, on the fly histogram computations and other image processing functions on the graphics hardware of the time. Recent developments in many/multi-core processors have resulted in the GPU and OpenGL becoming more and more suited to general purpose image processing tasks. Languages such as OpenGL's Shading Language (GLSL), the Compute Unified Device Architecture (CUDA) extensions to the C programming language and the Open Compute Language (OpenCL) now allow general purpose programmability of GPUs. This paper investigates the use of GPUs to accelerate the image registration implementation.

Block based image registration methods are the simplest. These compute correlation scores between the reference image and a set of candidate displaced images. A correlation score is typically calculated by taking the sum of absolute differences or sum of squared differences between the candidate displaced image and the reference image. Horn and Schunck [1981], and Lucas and

Kanada [1981] however presented optimised variational methods to compute optical flow that may be used to do image registration. The Lucas-Kanade (LK) algorithm is chosen for the current application for its known accuracy, robustness and suitability to a parallel implementation [Strzodka and Garbe 2004][Besnerais and Champagnat 2005][Marzat et al. 2009]. The LK algorithm of course operates under a brightness constancy assumption of the captured image stream, but this does not pose a problem at real-time frame rates during normal daylight video sequences.

The Lucas-Kanade image registration algorithm has previously been implemented on the GPU [Strzodka and Garbe 2004][Besnerais and Champagnat 2005]. Marzat, et al. [2009] also very recently did a GPU implementation using CUDA. The current paper however details the implementation of the LK algorithm in an existing performance critical surveillance system to do real-time 3D computational platform stabilisation. The performance result achieved by a pure GPU algorithm like that of Marzat, et al. is improved upon by implementing all algorithm steps for both the GPU and CPU. This is useful because not all the algorithm steps can be parallelised and the CPU is a more powerful sequential processor than the GPU. Therefore, when a system is deployed, the optimal CPU to GPU load balance may be selected for maximum performance of the stabilisation within the larger surveillance system.

Two regions of interest (see the greyed blocks in Figure 1) are registered in the high resolution wide angle image to allow very accurate registration under translation and rotation or to do 3D platform stabilisation as discussed in Section 4. To save on hardware resources an LK implementation is built that can register two input streams simultaneously. GLSL is used for the implementation because most of the existing surveillance system algorithms are implemented in GLSL. CUDA or OpenCL can however also be used if preferred.

1.2 Structure

As mentioned, this paper investigates the feasibility of a rigid image registration algorithm and parallel implementation that:

- Can flexibly balance the load between the CPU and GPU to optimally make use of the available resources, and
- would be efficient enough to enable 3D computational platform stabilisation in real-time.

Execution at 20 frames per second (fps) is considered real-time, but the algorithm will finally operate as part of a larger pipeline of processing steps.

The Lucas-Kanade image registration algorithm is discussed briefly in Section 2, followed by the details of the proposed GPU implementation in Section 3. The implications of applying the Lucas-Kanade implementation to do real-time 3D computational platform stabilisation, and the performance results in doing so, is given in Section 4 and Section 5 respectively. Some concluding remarks followed by recommendations for future research is finally given in Section 6.

2 The Lucas-Kanade Image Registration Algorithm

Image registration is the process of finding the transformation required on an input image to align it with some reference image. Further, rigid image registration implies that the transformation attempted does not warp the input image as shown in Figure 2. Registering from the left to the centre image is a rigid image transformation, but from the left to the rightmost image is a non-rigid transformation.

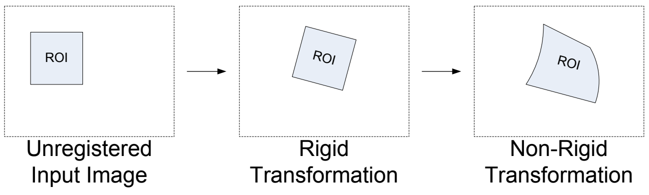
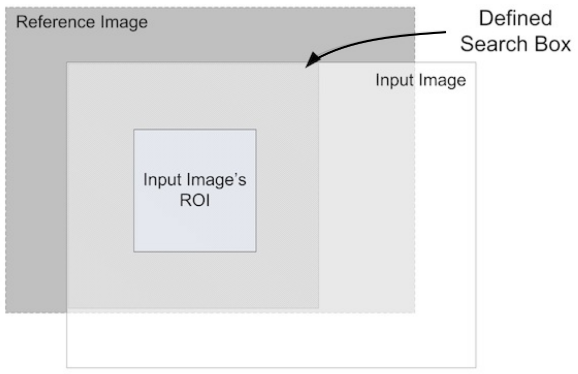


Figure 2: *Rigid vs. Non-Rigid Image Registration*

To introduce the Lucas-Kanade algorithm a brute force image registration algorithm is discussed first. The optimisations implemented by the Lucas-Kanade algorithm are then introduced.



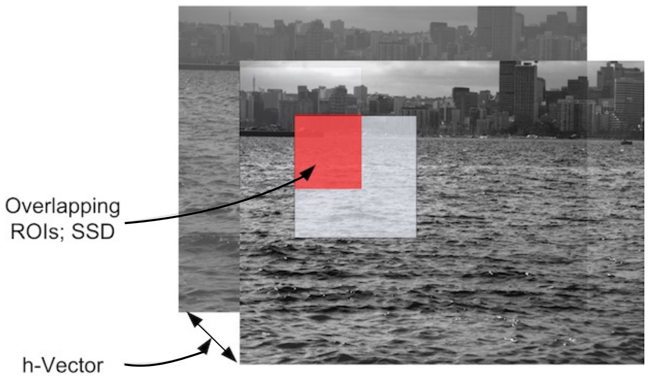


Figure 3: *Image Registration*

The Region of Interest (ROI) shown in Figure 3 refers to the area of the input image that the registration calculations are applied to. A search box around the ROI implements the maximum registration translation (up, down, left and right) considered when calculating the registration transformation. The registration vector (or h-vector) is the transformation that registers an input image to its reference image. One h-vector, and therefore one ROI, is required to register an image under translation.

Two spatially separated h-vectors, and therefore two spatially separated ROIs, are used to register an image under translation and very accurately under rotation. The two ROIs and their respective h-vectors set up a 2 dimensional referenced axis system within which rigid registration under translation and rotation is possible as shown in Figure 4. Two ROIs are also required for 3D platform stabilisation discussed in Section 4.

A brute force image registration algorithm attempts to minimise a

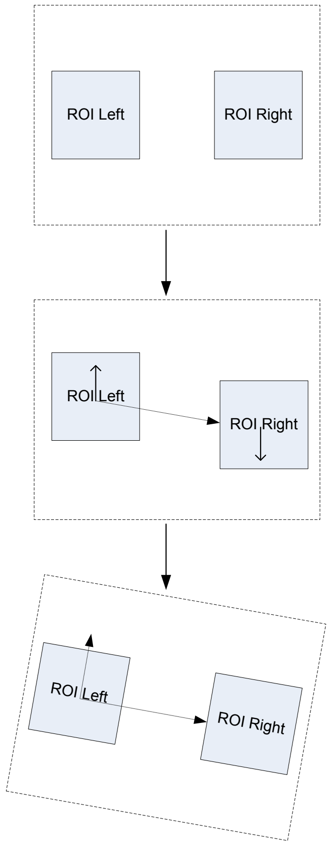


Figure 4: *Registration Under Rotation From Two Translations*

difference metric, usually the sum of squared differences or sum of absolute differences, within the ROI window. This is the simple block based method mentioned in the introduction. The registration is accomplished by computing the difference metric between the reference and input image for all candidate integer h-vectors within the predefined search box. The image registration operation then involves choosing the h-vector that minimises the difference metric.

Although such a brute force algorithm achieves pixel level accuracy, it is not very efficient and does not offer real-time performance on current hardware platforms. The first optimisation that the Lucas-Kanade algorithm introduces is to use a multi-resolution approach that finds the registration h-vector at a coarse detail level and then hierarchically refines it. The second optimisation is to find the best h-vector via a Newton-Raphson iterative optimisation instead of searching over the entire h-vector domain.

2.1 The Multi-Resolution Optimisation

In a multi-resolution approach the image registration operation (the h-vector search) is initially done on a low resolution version of the input and reference images. The result is then iteratively refined at consecutively higher resolution versions of the input and reference images until the highest resolution has been processed.

Imagine that the multiple resolutions of each image forms an image pyramid as shown in Figure 5. It is clear that the ROI in a lower resolution image contains less pixels than in a higher resolution image simply because there are less pixels in a low resolution image. For a native resolution ROI of 512×512 pixels and 6 levels in the pyramid, the lowest resolution ROI is only 16×16 pixels. More

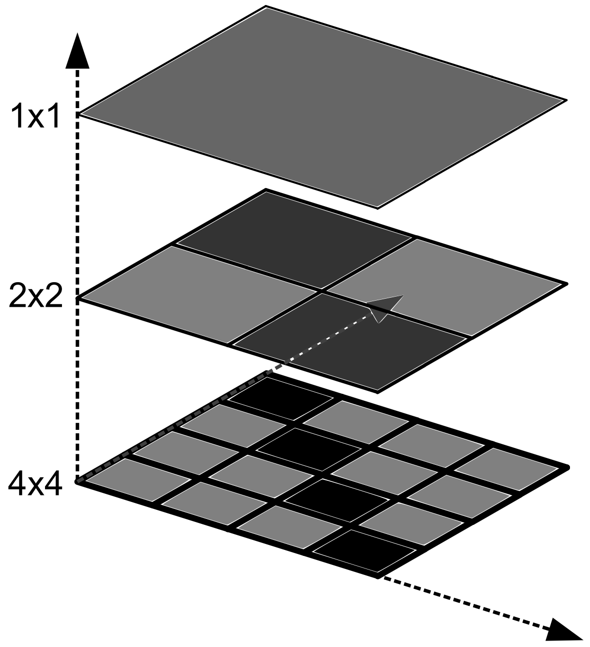


Figure 5: *Multi-Resolution Image Pyramid*

importantly, a low resolution search box of 3×3 pixels which requires only 9 search operations is equivalent to a maximum translation registration of (2^5+2^4+2^3+2^2+2^1+2^0) = 63 pixels in the native resolution camera image. That is a search domain of [-63, 63] pixels which is a search box of 127×127 pixels. By using multi-resolution search boxes one has a search space of $6 \times 3 \times 3 = 54$ potential h-vectors instead of a naive search space of $127 \times 127 = 16129$ potential h-vectors for the same effective search box. Additionally, the difference metric calculations are typically quicker to calculate at a lower resolution than at the higher resolution of the original images.

Such a multi-resolution algorithm however still contains a per-level-naive h-vector search. The h-vector search may be optimised further as discussed next.

2.2 Finding the h-Vector via a Newton-Raphson Iteration

Lucas and Kanade's algorithm simplifies the search problem under certain assumptions (discussed below) and then efficiently finds a solution of the h-vector via a Newton-Raphson iteration. The Lucas-Kanade algorithm implementation follows the intuitive 2D formulation presented by Lucas and Kanade [1981].

Firstly, a Gaussian filter kernel with a standard deviation of one is used to filter the input and reference images and to downsample the images when constructing the input and reference image pyramids. The per-pixel derivatives of the filtered images are also stored in a second and similar set of image pyramids.

The basic algorithm proceeds under the assumptions that

- The intensity of the filtered image $F(x, y)$ is linear near (x, y) which implies the derivative, $F'(x, y)$, is constant, and
- that there exists a small h-vector such that $F(x + h_x, y + h_y) = G(x, y)$ where G is the reference image.

Under these assumptions $h_{xy} \approx \frac{G(x,y)-F(x,y)}{F'(x,y)}$. The linearity assumption is enforced somewhat by the Gaussian filter kernel that is applied to the input and reference images and used to down-sample the images.

Note that an h-vector is calculated for each pixel in the image. Having an h-vector per pixel gives the algorithm its other name which is the Lucas-Kanade optical flow algorithm. Many computer vision processes use the optical flow information directly. For the purpose of rigid image registration the *average* h-vector over the ROI is however required.

The average h-vector is calculated by the weighted sum of the pixel h-vectors as derived from the minimisation of the sum of squared differences metric between F and G. Then,

$$h \approx \frac{\sum_{xy} F'(x,y)[G(x,y)-F(x,y)]}{\sum_{xy} F'(x,y)^2}.$$

However, because the linearity of $F(x,y)$ is only approximately true the h-vector is only approximate. A Newton-Raphson iterative solution is required to find the exact h-vector. Then,

$$h_{k+1} = h_k + \frac{\sum_{xy} F'\big(x+h_{k,x},y+h_{k,y}\big)\big[G(x,y)-F\big(x+h_{k,x},y+h_{k,y}\big)\big]}{\sum_{xy} F'\big(x+h_{k,x},y+h_{k,y}\big)^2}.$$

In a multi-resolution approach h_0 is initially set to zero, but the lower resolution estimate is propagated to each higher resolution level thereafter.

3 The Lucas-Kanade Algorithm Implementation

An Image Processing Framework (IPF) has been created by the CSIR's Optronic Sensor Systems group to facilitate the rapid development and deployment of image processing systems for the South African Armed Forces. The relevant functionality of the IPF is discussed briefly, followed by the proposed GPU+CPU implementation of the Lucas-Kanade algorithm.

3.1 The Image Processing Framework

The primary goal of the IPF is to enable image processing experts to focus on the development of algorithms and not on Input/Output (I/O) as is discussed by Delport [2009]. The framework therefore attempts to provide easy to use cross-platform data transfer from cameras or video files to the CPU and GPU.

Image registration needs to be performed in real-time on a high resolution, wide angle image for the wide area surveillance demonstrator. The wide angle image is created by stitching video received from multiple Gigabit Ethernet digital cameras. It is important for both the stitching and registration algorithms that the images from the cameras are synchronised and delivered for processing in a timely manner. The framework delivers the required functionality by being optimised for multithreaded high bandwidth acquisition.

Internally, the framework makes use of the open source FFmpeg and OpenSceneGraph (OSG) libraries. FFmpeg is used for reading and writing a wide variety of video files, while OSG is built on top of OpenGL and allows for implementing image processing algorithms in GLSL and CUDA. Having the processed image data in to GPU also allows it to be displayed using OpenGL; linear filtering, panning and zooming can therefore easily be performed.

Framework users can write image processing stages in either of the following:

- C/C++ code that utilises OpenMP to execute on all the cores of a multi-core CPU.
- GLSL or CUDA code that executes on the many-core GPU.

The processing stages can then be chained together into a processing graph.

3.2 The Lucas-Kanade GPU Implementation

The basic building block of the process graph is shown in Figure 6. The Lucas-Kanade root node of the graph controls the execution order of the processing steps attached to it. The execution order is usually from left to right.

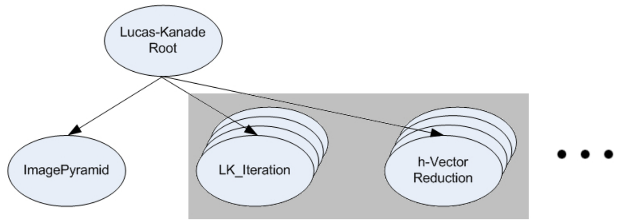


Figure 6: *Building Block of Process Graph for the Lucas-Kanade Algorithm*

The leftmost node in Figure 6, i.e. the ImagePyramid node, firstly constructs a multi-resolution image pyramid. Each LK_Iteration node, shown stacked on top of each other, then does one Newton-Raphson iteration. There are typically seven LK_Iteration nodes attached to do seven Newton-Raphson iterations. Indeed Marzat, et al. [2009] have shown that the optical flow error levels off beyond six to seven iterations. Finally the stack of h-Vector reduction nodes performs the weighted sums required to hierarchically reduce the output from the LK_Iteration to a single h-vector; the CPU does this sequential operation in a single pass while each GPU h-vector reduction pass only reduces the image resolution by a factor of two. The number of GPU h-vector reduction passes—before handing the final weighted sum over to the CPU—may be adjusted for best performance on the target hardware. It should be noted that although only one LK_iteration stack and one reduction stack is shown that a pair of LK_iteration and reduction stacks is required for each level in the image pyramid.

As mentioned, the LK_Iteration step uses an image pyramid of the current input image as well as an image pyramid of the reference image. The reference image is updated initially and then once in a while as required. To register the input image under rotation or do 3D platform stabilisation a left and a right ROI image pyramid is used for each of the input and reference images. Four image pyramids are therefore required in total.

The process graph for the full Lucas-Kanade algorithm is shown in Figure 7. The (A) and (B) sub-graphs respectively represent the input and reference image pyramids. The osgSwitch nodes allow the input and reference pyramids to be updated individually. If, for example, switch A is closed and B is open then only the input image pyramid would be updated while the reference image and pyramid is maintained. Remember that only one pair of LK_iteration and reduction stacks are shown per branch, but that such a pair is in fact added for each level in the pyramid.

For performance reasons the entire process graph is constructed at initialisation and not modified during run-time. This however has the drawback that the GPU implementation just described was limited to a ROI size of 256x256 under Linux. This was due to a

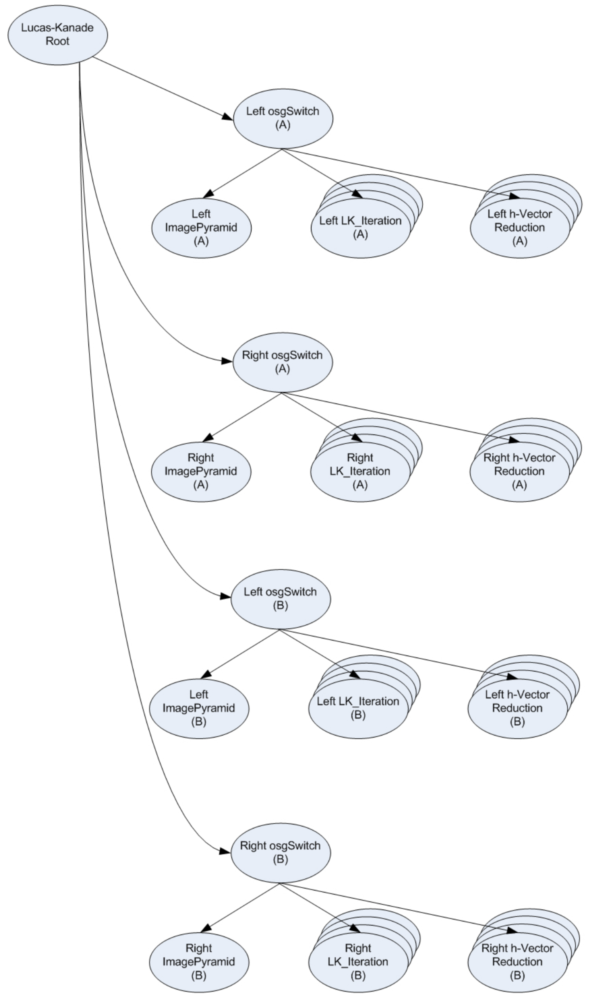


Figure 7: *Full Process Graph for the Lucas-Kanade Algorithm*

OpenGL driver issue that limited to 200 the number of cameras and therefore the number of processing passes that could simultaneously be in a single process graph. Such a small ROI limits the search box size and therefore the h-vector to a maximum of 31 pixels under translation. Such a limited registration distance prevents registration of camera motion larger than 31 pixels per frame.

To save on algorithm passes, the left and right ROI pyramids may however be executed simultaneously (within one pass) instead of doing them serially one after the other. Only two sub-graphs are therefore attached to the Lucas-Kanade root: One for the left and right input ROI pyramids and a second for the left and right reference ROI pyramids. The h-vector reduction pass therefore also operates on two images simultaneously.

A further saving on the number of camera passes is achieved by sharing the per pyramid level h-vector reduction pass between the input and reference sub-graphs. The modified process graph of the full algorithm is shown in Figure 8. Again the (A) and (B) subgraphs are for the input and reference images respectively. The composite algorithm that registers two ROIs simultaneously is referred to as a bi-Lucas-Kanade algorithm in the rest of the paper. Note that the image pyramid building pass and the LK_Iteration pass use different image sources in (A) and (B). These passes are therefore not currently shared between the input (A) and reference (B) sub-graphs. Remember that only one pair of LK_iteration and

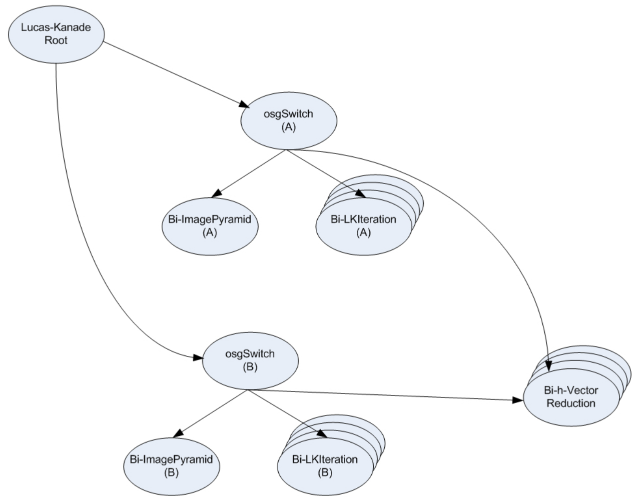


Figure 8: *Optimised Process Graph for the full Lucas-Kanade Algorithm*

reduction stacks are shown per branch, but that such a process pair is in fact added for each level in the pyramid.

3.3 The Reference CPU Implementation

The process graph abstraction provided by the IPF allows an algorithm pass (such as the Newton-Raphson iteration) to be implemented either on the CPU or on the GPU. If both a CPU and a GPU implementation of an algorithm pass exists then it allows one to flexibly balance the work load between the GPU and CPU. The CPU load, the GPU load and the communication bandwidth between the CPU and GPU [1] is optimised by running parts of the algorithm on the CPU and parts on the GPU. Such load balancing is often required for best performance on the target hardware and within the larger set of image processing steps in the system.

All algorithm passes of the Lucas-Kanade algorithm were implemented both on the CPU and the GPU. The CPU algorithm passes were in fact implemented first to serve as a validation and performance reference.

4 Computational Platform Stabilisation

The aim of this paper is to show how real-time computational platform stabilisation may be done by making use of an accelerated implementation of an image registration technique. To perform the computational platform stabilisation the image registration results are used to estimate the 3D pose of the surveillance platform relative to either the previous frame or, via integration, some earth-fixed or initial orientation state. Specifically the Euler rotation matrix which aligns the rotated axis system (X', Y' and Z') in Figure 9 to the chosen reference axis system (X, Y, and Z) is desired. This matrix can then be fed back to the mechanical systems to perform the stabilisation or be used to warp the wide angle image to implement computational (i.e. virtual) platform stabilisation.

The mathematical notation used in this section is as follows: A 3-dimensional vector, V_{abc}, is a vector from point b in the direction of point a expressed in terms of its projection onto orthogonal

[1]The 4 GB/s PCI Express bus bandwidth between the GPU and CPU is rather limited when compared to the 20+ GB/s bandwidth from CPU to system RAM and 220+ GB/s bandwidth from GPU to device RAM.

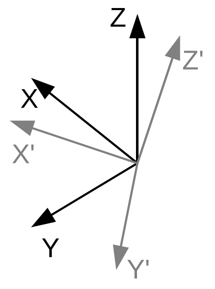


Figure 9: *Rotated Axes*

coordinate system c. V_{abc} is used when the magnitude of the vector is unknown or unimportant. T_{abc} represents the translation or displacement of point a relative to point b. U_{abc} is a unit vector pointing in the direction of point b to point a. R_{ab} is a 3x3 Euler rotation matrix expressing the rotation of orthogonal axis system a relative to (and in terms of its projections on) orthogonal axis system b. Individual elements of 3 dimensional vectors are referred to as x, y, or z whereas the elements of the two dimensional vectors are referred to as horizontal (h) and vertical (v) to avoid confusion.

This paper assumes that the image has been spherically stitched [de Villiers 2009], although this only affects the pixel to vector (1) transformation. It is however still necessary to determine the distortion parameters [de Villiers et al. 2008] and focal length [de Villiers 2009] if using a single camera, so that proper conversion from pixel space to the undistorted image plane can be done.

In the axis system used in this paper, the origin is located at the focal/convergence point of the lens or centre of the spherical stitch. The X-axis is coincident with the optical axis and is positive in the direction the camera is looking. The Y-axis is positive towards the left of the image, and the Z-axis positive upwards. Positive yaw is clockwise rotation when looking in the positive Z direction, positive pitch is anticlockwise when looking in the positive Y direction, and positive roll is clockwise when looking in the positive X direction. Figure 10 displays the axis definitions, in Figure 10(a) the Z axis comes out the page, and in Figure 10(b) the Y axis comes out the page.

To determine the required Euler rotation to align two successive spherically stitched images, only the coordinates of two image points are required in each frame assuming that the camera parameters remain constant and the points are static relative to each other in the scene being viewed. The first step is to convert the two tracked coordinates for frame n and frame $n+1$ into four vectors using (1).

$$U = \begin{bmatrix} cos((C_v - P_v)/R_v) * cos((C_h - P_h)/R_h) \\ cos((C_v - P_v)/R_v) * sin((C_h - P_h)/R_h) \\ sin((C_v - P_v)/R_v) \end{bmatrix} \tag{1}$$

where:

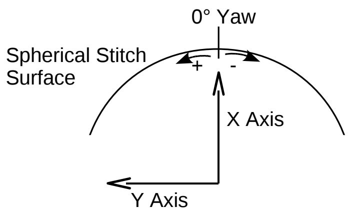
$(C_h, C_v) =$ coords of the centre of the stitch

$(P_h, P_v) =$ pixel position

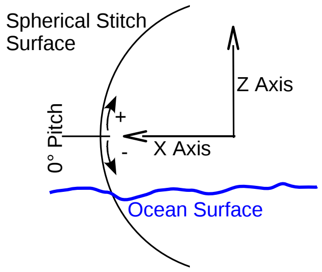
$(R_h, R_v) =$ angular resolution of stitch in pixels/rad,

$U =$ the unit vector corresponding to the pixel P.

It is possible to create a local orthogonal vector space from 3 points



(a) XY Plane Cross Section



(b) XZ Plane Cross Section

Figure 10: *Spherical Axis System Definition*

as shown by (2). The first point is deemed the origin of the vector space, and the X axis is the unit vector pointing from the first to the second point. The third point definess the XY plane and is used to calculate Z axis, which then is used to calculate the Y axis.

$$\begin{aligned} X &= \frac{B - A}{\parallel B - A \parallel} \\ Y' &= \frac{C - A}{\parallel C - A \parallel} \\ Z &= X \otimes Y' \\ Y &= Z \otimes X \end{aligned} \tag{2}$$

$$R = \begin{bmatrix} X_x & Y_x & Z_x \\ X_y & Y_y & Z_y \\ X_z & Y_z & Z_z \end{bmatrix} \tag{3}$$

where:

$A =$ first input 3D point,

$B =$ second input 3D point,

$C =$ third input 3D point,

$Y' =$ temp vector in XY plane,

$\otimes =$ vector cross product,

$X =$ X Axis vector of local coord system,

$Y =$ Y Axis vector of local coord system,

$Z =$ Z Axis vector of local coord system, and

$R =$ Euler rotation of local coord system relative to the axis system in which the points were expressed.

To calculate the Euler transformation using the two tracked points, a local coordinate system is created using the vectors created from the two tracked points and the spherical stitch's reference, here assumed normalised to $(0, 0, 0)$. Using these two matrices the rotation from frame n to frame $n + 1$ can be calculated but the result is expressed in the local coordinate system of frame n and must first be converted back to the stitch's reference frame before it is useful.

$$\begin{aligned} R' &= R_n^T \times R_{n+1} \\ R_\Delta &= R_n \times R' \times R_n^T \end{aligned} \tag{4}$$

where:

R_n = rotation matrix created from (2), using (0,0,0) as vec A, and the the two tracked points' vectors in frame n for B and C

R_{n+1} = rotation matrix created from (2), using (0,0,0) as vec A, and the the two tracked points' vectors in frame n for B and C

R' = Euler rotation from frame n to frame $n + 1$, expressed in R_n's axis system, and

R_Δ = Euler rotation from frame n to frame $n + 1$, expressed in the stitch's axis system.

The bi-Lucas-Kanade image registration implementation described in the previous sections may therefore be used to register an image from a single camera under translation and rotation or it may be used as described above to do real-time 3D computational platform stabilisation.

5 Results

A test pattern was created to analyse the bi-Lucas-Kanade algorithm implementations during development. The checkerboard test pattern is shown in Figure 11. Notice that the checkerboard patterns have been registered against a page-aligned checkerboard. To ensure a unique registration solution, the checkerboard elements of course have to be larger than the largest pixel motion expected per frame.

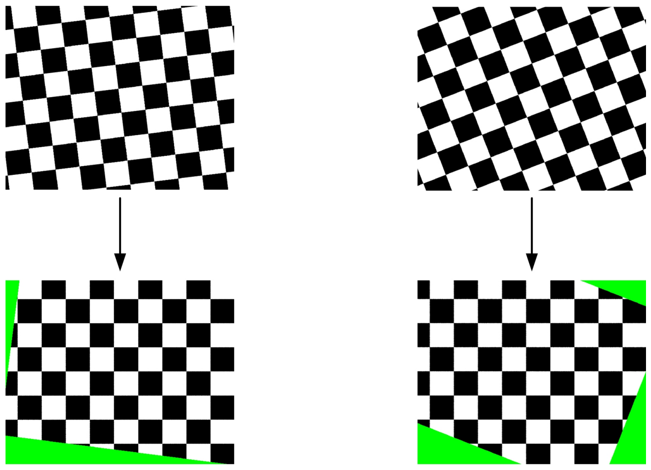


Figure 11: *Registered/Stabilised Checkerboard Patterns*

Note that because the number of Newton-Raphson iterations are fixed to seven that the time complexity of the Lucas-Kanade algorithm and the performance results of the implementation is independent of the image sequence used. The performance results were generated on a PC with an Intel Core 2 Quad 2.83GHz CPU (only one core was used to generate results) and an NVidia GeForce GTX285 GPU. The computational load between the CPU and the GPU may be balanced in 3 ways:

- The image pyramid construction may be done on the GPU or CPU,
- the Lucas-Kanade iterations may be done on the GPU or CPU irrespective of where the image pyramids are constructed, and
- the number of h-vector reduction passes (partial sums) done on the GPU may be varied from zero to many if the Lucas-Kanade iterations are done on the GPU.

Table 1 indicates the performance of the Lucas-Kanade algorithm running in isolation. The stitched wide angle image stream has a resolution of, for example 4096×1000 pixels, but the bi-registration is done on two 256×256 or 512×512 ROIs. The limitation on the number of process graph cameras (and therefore on the number of algorithm passes) described earlier prevents more than 2 GPU h-vector reduction passes for ROIs of $512 \times 512 \times 2$.

CPU vs. GPU Load	$256 \times 256 \times 2$ ROI	$512 \times 512 \times 2$ ROI
CPU Only	10 fps	2.6 fps
GPU Pyramid Construction	39 fps	10 fps
GPU Pyramid+LK Iterations	46 fps	15 fps
1 GPU h-vector reduction	50 fps	30 fps
2 GPU h-vector reductions	44 fps	30 fps
3 GPU h-vector reductions	38 fps	-

Table 1: *CPU vs. GPU Performance Results*

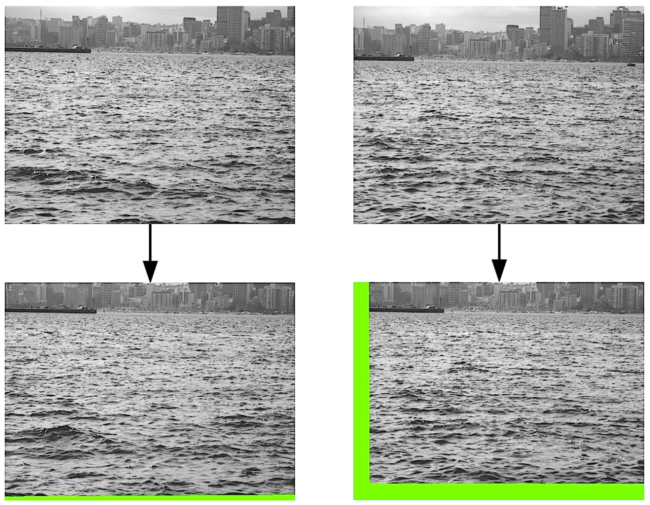


Figure 12: *Maritime Stabilisation Results*

The GPU+CPU implementation achieves a maximum measured performance of $512 \times 512 \times 2 \times 30fps = 15728640$ pixels per second. From the table it is clear that the best CPU to GPU load balance is when the pyramid construction, the LK iterations and one h-vector reduction pass is done on the GPU. Figure 12 shows two images from the stabilisation sequence that generated the above results. As shown in the table, the ROI size has an impact on the implementation's performance.

When the same set of experiments are run on a Macbook Pro 13" laptop containing an NVidia Geforce 9400M GPU and a 2.26GHz Core 2 Duo CPU one finds a maximum frame rate of around 8 fps and 5 fps for ROIs of 256×256 and 512×512. For this platform however the maximum frame rate is only achieved when all the algorithm steps are ran on the GPU.

For the stabilised platform, the input image is the latest image from the camera and the reference image is the previous image from the camera. For this use case, seven Newton-Raphson iterations are enough for an average registration accuracy of 0.024 pixels per frame. When the LK iterations are done on the CPU an average registration accuracy of 8×10^{-8} pixels per frame is achieved.

With their recent CUDA implementation Marzat, et al. [2009] achieves a frame rate of 15 fps at an image and ROI size of 640×480 pixels. This result was measured on an NVidia Tesla C870 GPU with about half as many processor cores as the GeForce GTX285 GPU. Marzat, et al. however estimate that their execution times would half with double the number of processors i.e. 9216000 pixels per second ($640 \times 480 \times 30 fps$) on the Geforce GTX hardware.

6 Conclusion

6.1 Discussion

The GPU implementation of the Lukas-Kanade algorithm is more efficient than a GPU multi-res brute force algorithm would be if the Newton-Raphson iterations converge in less than $5 \times 5 = 25$ iterations. This does seem to be the case with 7 Newton-Raphson iterations typically being required. Additionally the Lucas-Kanade algorithm already gives sub-pixel accuracy while the brute force approach does not.

The GPU implementation of the Lucas-Kanade algorithm achieved real-time performance for ROIs of 256×256 and 512×512 pixels. Both the pyramid construction and the LK iteration passes seem to be more suited to the GPU implementation than the CPU implementation. One GPU h-vector reduction increases the performance significantly. Additional GPU h-vector reduction passes however decrease the performance or provide little benefit due to the added overhead of more render passes.

It has therefore been shown that using both the CPU and GPU could enhance the implementation's performance over that of a pure GPU based implementation. The performance of this implementation also compares very well to the performance of the recent CUDA implementation by Marzat, et al. [2009].

6.2 Future Work

Manually positioning the ROIs over parts of the image that contain a stable backdrop with spatial detail is currently operationally unavoidable. Pre-selecting an area in the image that will always contain a stable backdrop would not be possible. It is therefore proposed to also investigate feature based stabilisation. A possible algorithm could be based on the Kanade-Lucas-Tomasi (KLT)-tracker [Tomasi and Kanade 2009]. Feature based stabilisation potentially has the advantage of being able to dynamically auto-select the parts of the image that contain a stable backdrop.

It would also be interesting to look at other stabilisation methods. For example, an inertial measurement unit (IMU) has already been added to the WASP. It would be interesting to use the IMU to do a first order stabilisation while the LK stabilisation does the higher order corrections which would require only a relatively small search space.

Acknowledgements

The authors would like to thank The Armaments Corporation of South Africa (Armscor) for funding this work.

References

BESNERAIS, G. L., AND CHAMPAGNAT, F. 2005. Dense optical flow by iterative local window registration. In *In IEEE International Conference on Image Processing 2005*, I – 137–40.

DE VILLIERS, J. P., LEUSCHNER, F. W., AND GELDENHUYS, R. 2008. Centi-pixel accurate real-time inverse distortion correction. In *Proceedings of the 2008 International Symposium on Optomechatronic Technologies*, vol. 7266 of *ISOT2008*, 1–8.

DE VILLIERS, J. P. 2009. Real-time photogrammetric stitching of high resolution video on COTS hardware. In *Proceedings of the 2009 International Symposium on Optomechatronic Technologies*, vol. 9 of *ISOT2009*, 46–51.

DELPORT, J. P. 2009. Optronics image processing framework position paper. Tech. rep., CSIR. Position Paper.

FLETCHER, R., AND REEVES, C. M. 1964. Function minimization by conjugate gradients. *Computer Journal 7*, 140–054.

HORNE, B. K. P., AND SCHUNCK, B. G. 1981. Determining optical flow. *Artificial Intelligence 17*, 185–203.

JEEBODH, P., AND DUVENHAGE, B. 2008. Super resolution. Tech. Rep. 6700-OPTO-38601-01, CSIR. Restricted document.

KAHN, F., CHAPMAN, M., AND LI, J. 2005. Camera calibration for a robust omni-directional photogrammetry system. In *Proceedings of the 5th International Symposium on Mobile Mapping Technology*, MMT07, 1–8.

LUCAS, B., AND KANADE, T. 1981. An iterative image registration technique with application to stereo vision. *International Journal on Computer Vision and Image Processing*, 674–679.

LUCCHESE, L., AND MIRA, S. K. 2002. Using saddle points for subpixel feature detection in camera calibration targets. In *Proceedings of the Asia-Pacific Conference on Circuits and Systems*, vol. 2, 191–195.

MA, W., 2007. Riding shotgun with google street views revolutionary camera. http://www.popularmechanics.com/-technology/industry/4232286.html?page=2.

MARZAT, J., DUMORTIER, Y., AND DUCROT, A. 2009. Real-time dense and accurate parallel optical flow using CUDA. In *Proceedings of The 17th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, WSCG2009, 105–111.

OWENS, J. D., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J. E., AND PHILIPS, J. 2008. Gpu computing. *Proceedings of the IEEE 96*, 5, 879–899.

SNYMAN, J. A. 1983. An improved version of the original leap-frog dynamic method for unconstrained minimization: LFOP1(b). *Applied Mathematics and Modelling 7*, 216–218.

STRZODKA, R., AND GARBE, C. 2004. Real-time motion estimation and visualization on graphics cards. In *Proceedings IEEE Visualization 2004*, 545–552.

TOMASI, C., AND KANADE, T. 2009. Detection and tracking of point features. Tech. Rep. CMU-CS-91-132, Carnegie Mellon University.