Methods for 3D Structured Light Sensor Calibration and GPU Accelerated Colormap

METHODS FOR 3D STRUCTURED LIGHT SENSOR CALIBRATION AND GPU ACCELERATED COLORMAP

BY

VENU KURELLA, M.Sc., (Mathematics)

University of British Columbia, Vancouver, Canada

A THESIS

SUBMITTED TO THE SCHOOL OF COMPUTER SCIENCE & ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF McMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Doctor of Philosophy

Doctor of Philosophy (2017) McMaster University

(School of Computational Science and Engineering) Hamilton, Ontario, Canada

TITLE:	Methods for 3D Structured Light Sensor Calibration and GPU Accelerated Colormap
AUTHOR:	Venu Kurella PhD student
SUPERVISORS:	Dr. Allan Spence and Dr. Christopher Anand
NUMBER OF PAGES:	xvi, 109

To the love and support of family, friends and well-wishers whose brought me in tolerance with the PhD requirements.

Abstract

In manufacturing, metrological inspection is a time-consuming process. The higher the required precision in inspection, the longer the inspection time. This is due to both slow devices that collect measurement data and slow computational methods that process the data. The goal of this work is to propose methods to speed up some of these processes. Conventional measurement devices like Coordinate Measuring Machines (CMMs) have high precision but low measurement speed while new digitizer technologies have high speed but low precision. Using these devices in synergy gives a significant improvement in the measurement speed without loss of precision. The method of synergistic integration of an advanced digitizer with a CMM is discussed. Computational aspects of the inspection process are addressed next. Once a part is measured, measurement data is compared against its model to check for tolerances. This comparison is a time-consuming process on conventional CPUs. We developed and benchmarked some GPU accelerations. Finally, naïve data fitting methods can produce misleading results in cases with non-uniform data. Weighted total least-squares methods can compensate for non-uniformity. We show how they can be accelerated with GPUs, using plane fitting as an example.

Acknowledgements

I am grateful and fortunate to have Dr. Allan Spence and Dr. Christopher Anand as my supervisors. Their guidance, support and encouragement helped me navigate not just my PhD but also my life. I am thankful to Drs. Veldhuis, Fleisig, Bone for the guidance throughout the program. Special thanks to Dr. Qiao and Dr. Mayer for their valuable feedback on my thesis. I appreciate the financial support provided through NSERC and Origin International.

I would also like to thank Kai for being such a great colleague. I must acknowledge the invaluable company of Hamad, Graham, Yahu, Mohamed, Behrad, Yile, Rong, Yangliu and Cris. Thanks go to Bob and Murray of Origin and Ron, Mark, John, Joe and Dan of machining lab for their kind and patient support during my research. Special appreciation goes to Bartek and Julie of CSE and Dr. Lightstone, Lily, Florence and Vania of mechanical Engineering for going above and beyond to help me out. I must also mention the valuable mentorship of Steve and Jeremy.

I am thankful to my loving wife, Parathy, and friends, Jessie, Sravan, Vishnu, Sathish and Naresh for their unwavering love, support and encouragement. Finally, I would like to express my gratitude to my parents, Laxmi and Kumaraswamy, sister, Swetha and brother, Praveen, for their unending support and love throughout my life.

Notation and Abbreviations

3D	Three Dimensional
CAD	Computer Aided Design
CAM	Computer Aided Manufacturing
CMM	Coordinate Measuring Machine
DLL	Dynamic Link Library
FFM	Facet Facet Matching
GD&T	Geometric Dimensioning and Tolerancing
GPGPU	General Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
HTM	Homogeneous Transformation Matrix
LED	Light Emitting Diode
NIST	National Institute of Standards and Technology
OLS	Orthogonal Least Squares
PFM	Point Facet Matching
TLS	Total Least Squares
WTLS	Weighted Total Least Squares

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Metrology is the study of measurement. In manufacturing, metrology relates to inspection of manufactured parts to ensure that they adhere to the dimensions, within given tolerances, set by the design. In industries like automotive manufacturing and aerospace, metrology is a crucial but very time-consuming process. This work draws motivation from the need to speed up the inspection process. Metrological inspection of a part can be broken down into two stages: measurement with precise devices and processing the measurement data to check for tolerances. This work proposes methods to speed up these two stages.

Many parts manufactured in the automotive industry have very tight tolerances requiring micrometer level measurement precision. CMMs are the most reliable measuring devices in automotive industry. They have been widely used for decades. While most CMMs have micrometer level precision, they are known to be slow. Digital measurement devices that arrived in the past few years are faster than a CMM but not as precise; hence, researchers have come up with methods to integrate digital devices with CMM to get an intermediate speed without any loss of precision. Chapter 2 of

this work discusses a novel integration of a state-of-the-art digital sensor and a CMM. It is an expanded version of the paper [64].

In the second stage of metrological inspection, collected measurement data is compared against the CAD model to check if the part is manufactured within tolerances. Several computational methods exist to perform this comparison. They use the points from the data and facets from the CAD model for this purpose. Deviations are calculated for every part feature and compared against the tolerances in its design. Only if the deviations are within the tolerances, does the part pass the inspection. Comparison software tools have been widely implemented on conventional CPUs. While the CPU implementations are suitable for small data sets, *e.g.* those from CMMs, digitizers produce dense data making it a time-consuming process. Recent GPU technologies have been used to accelerate the CPU implementations. GPUs are also inexpensive. In chapter 3, we study the acceleration methods in the context of an industrial problem. The results have been summarized in the publication [37].

Existing point cloud to model comparison tools do not account for non-uniformity in point clouds. Weighted total least squares have been developed by researchers at NIST to tackle the basic problem of plane fitting. We show how this method can benefit from GPU acceleration, in chapter 4 which is also submitted for publication. Each chapter sets the stage with a review of the relevant literature. The last chapter is summary of contributions of the thesis work.

Chapter 2

Calibration of a Compact Snapshot Sensor

2.1 Digital Sensors in Metrology

Coordinate Measuring Machines (CMMs) are a broad category of machines that perform Geometric Dimensioning and Tolerancing (GD&T) based measurement of parts. A CMM reports points on a part by making contact with a touch probe. There are two major types of CMMs: Cartesian CMMs (Fig. 2.1a) which give Cartesian (XYZ) coordinates directly and non-Cartesian CMMs (Fig. 2.1b) which use angles along with translations to give the required coordinates [25].

While each have their own applications, this work focusses is Cartesian CMMs that are by far the most prevalent, accurate and well-investigated among the two types. Conventionally, touch probes are used with CMMs in geometric measurement. Touch or tactile probes touch a part to report the XYZ coordinates of the point of contact. These are the most precise among the methods of measurement in manufacturing.

(a) IOTA-P® CMM (b) FaroArm® [17] (c) Renishaw ® probe [54]

Figure 2.1: Conventional metrology: (a) Cartesian CMM (b) FaroArm articulated arm CMM (c) Touch probe reporting the X, Y and Z coordinates of the point of contact

However, they have two major limitations: slow point collection capability (up to a few hundred points per second) and the need for making contact with the part for measurement - a concern when a part could be deflected due to touch. Because of these reasons, over the past few years, touch probing on CMMs has been augmented by the incorporation of 3D digitizers. These digital sensors have high speed, high resolution and are able to perform contactless measurement. Manufacturing industry has heavily invested in these technologies in the past few decades.

2.2 Types of Digital Sensors

Currently there are various types of digitizers used in metrology in the manufacturing industry. Their usage generally depends on the type of application and the required accuracy.

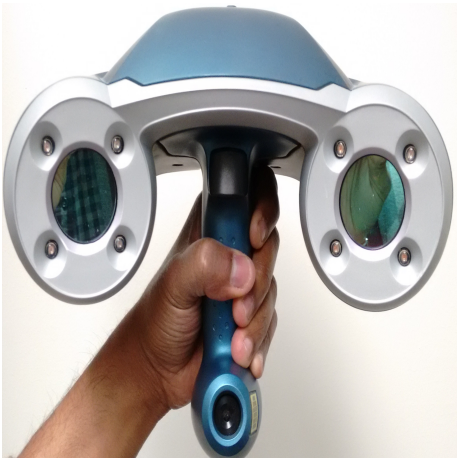
i. *Structured light sensors* use triangulation or fringe based techniques to calculate the distance of a part from the sensor (Fig. 2.2b). Laser line scanners and visible light sensors, including fringe based sensors, come under this category. These are the most accurate among the digitisers. They find applications not only in the manufacturing industry for part inspection and reverse engineering but also in other fields such as plastic surgery and dentistry [34].

ii. *Computed Tomography (CT)* uses ionizing radiation passing through a part to obtain its image. For manufacturing purposes, CT is discouraged since it is expensive, hard to calibrate and relatively less accurate.

iii. *Stereo vision digitizers* extract information from images of a part taken from multiple views (Fig. 2.2a). Compared to structured light sensors, stereo vision sensors have lower accuracy.

Barbero *et. al.* [6], study and compare three different kinds of laser scanners (including one mounted on a CMM), a CT method and a fringe projection method. They study parameters like accuracy, part digitization, distribution of scan points, mesh edges, roughness of meshing and holes without meshing. This work concluded that a laser scanner mounted on a CMM and fringe based sensors have better accuracy compared to other techniques. Triangulation based laser scanners and fringe-based

(a) Stereo sensor [59]



(b) Creaform ® scanner

Figure 2.2: Stereo and triangulation sensors

projection digitizers are used in 90% of the close-range digitizing applications in industry [27]. A Fiat Chrysler Automobiles technical report discusses investment in the digitizing techniques leading to decline of problems in vehicles [47]. While scanners are commercially available, they can also be custom built to suit specific needs [48, 50].

2.2.1 Structured Light Sensors

In metrology, widely used structured light sensors are of two types: laser line scanners and fringe based sensors. Line scanners project a beam of light within a 2D field of view to collect 3D data along the line of incidence. Fringe sensors, on the other hand, project patterns of visible light to obtain 3D data within the 3D field of view. The difference is illustrated in the documentation of Gocator® sensors from LMI Technologies Inc. [21] (Fig. 2.3).

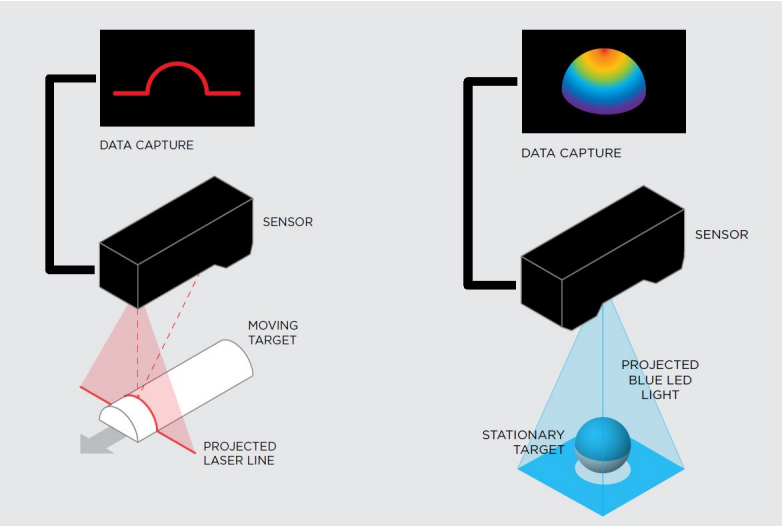


Figure 2.3: Gocator® laser line scanners and fringe sensor differences as shown in their documentation [21]

2.2.2 Laser Line Scanners

Commercially available laser scanners are of two kinds: mountable and portable scanners. While the former can be mounted on a CMM or a robotic arm, portable scanner is generally hand-held. The advantage of portable scanners is that they can be used to get a quick 3D point cloud of an object. The drawbacks are that their accuracy is generally lower than that of the mountable scanners, and they usually require a person to move and scan around the part.

2.2.3 Fringe Sensors

Fringe based sensors, like laser line scanners, can also be placed into two categories: stationary sensors and mountable sensors. Stationary sensors can be either purchased or custom built using visible light projectors [52]. Mountable sensors have been a recent development. Compared to laser scanners, they have been less prevalent

in manufacturing due to lower precision and greater size, making them difficult to mount on CMMs. With recent advances in technology, fringe sensors have come closer to scanners in precision and size. LMI Technologies Inc. developed a fringe based sensor [21] called a snapshot sensor. This sensor can be mounted on a CMM due to its light weight and compact size and is capable of sheet metal precision ($\sim$0.1 mm).

While both laser scanners and fringe sensors have comparable speed and precision, laser scanners are not considered eye-safe due to the focussed laser beam. Fringe sensors, on the other hand, avoid this problem. Both kinds of mountable sensors can be automated for manufacturing inspections on assembly lines. Once mounted on a CMM, they have to be calibrated.

2.3 Calibration

In metrology, calibration means determining the transformation between two frames. A calibration artefact is used for this purpose.

2.3.1 Calibration Artefact

A calibration artefact is a part with known geometry and dimensions used to calibrate measurement devices. The artefact is measured by devices to obtain their position and orientation with respect to the artefact. This information can be used to establish a transformation between the devices' frames. The artefacts are usually stationary, resulting in a consistent measurement frame. Once a calibration artefact is designed, it is measured by the devices. Mathematical methods are applied to the

measurement data to estimate the devices' position and orientation. These mathematical methods are developed using Homogeneous Transformation Matrices (HTMs) and feature fitting algorithms discussed in Sec. 2.5.

2.3.2 Calibration Types

Calibration is of two kinds: intrinsic and extrinsic [12, 65]. A structured light sensor has a camera that takes images of laser lines or fringe patterns to calculate X, Y, Z coordinates using triangulation or stereo methods. Intrinsic calibration determines this transformation between the image plane and the internal sensor coordinate frame based on its camera parameters. During digitizer manufacture, intrinsic calibration is a standard procedure where artefacts like checker board patterns and grids are used [52, 65]. When sensors are mounted on external devices like CMMs, the transformation between the CMM and the sensor frames has to be determined. This process, known as extrinsic calibration, helps establish a common coordinate frame to represent data obtained by touch probing or scanning a part. The focus of this work will be extrinsic calibration.

2.3.3 Extrinsic Calibration

Extrinsic calibration is the correction for small mounting misalignments when mounted on an external machine like a CMM on the shop floor. This is required to use the multi-sensor system in a synergistic way. It is performed in two stages: mechanical and mathematical. First, the roll, pitch and yaw angles are adjusted mechanically using a Renishaw® adjustment module to manually align the axes of the frames as much as possible [53] (Fig. 2.4). Then they are mathematically compensated using

a precise calibration method. The mathematical compensation requires a calibration artefact.

Figure 2.4: Renishaw® AM1 module [53]

2.3.4 Existing Calibration Methods

Mathematical calibration, referred to as just calibration in the rest of the thesis, was performed by various researchers on CMM mounted laser scanners. Calibration methods for commercial laser scanners are proprietary and are usually not discussed in the literature. In the academic literature, spheres are used for extrinsically calibrating custom built scanners [12]. The literature on fringe sensors focuses on intrinsic calibration, since stationary sensors are more prevalent than mountable ones [7, 10, 52]. To the best of our knowledge, there was no known extrinsic calibration method for fringe sensors available in literature. The main contribution of this work is an extrinsic calibration method for the CMM mounted snapshot sensor.

2.4 Snapshot Sensor

The snapshot sensor, powered by blue LED structured light, can take a snapshot of a sizeable 3D zone within its field of view [21](Fig. 2.5) and provides XYZ coordinates of the points in the region. The snapshot sensor uses a combination of triangulation and stereo methods to obtain the 3D information of a part. To the best of our knowledge, at the time of purchase, it was the only 3D structured light sensor with sheet metal precision that was compact and light enough to be mounted on a CMM. This gives it the advantage of quickly measuring large parts (especially sheet metal) on the shop floor. It can also be used to give the CMM an approximate location of features on a part that deviates from its CAD/nominal geometry.

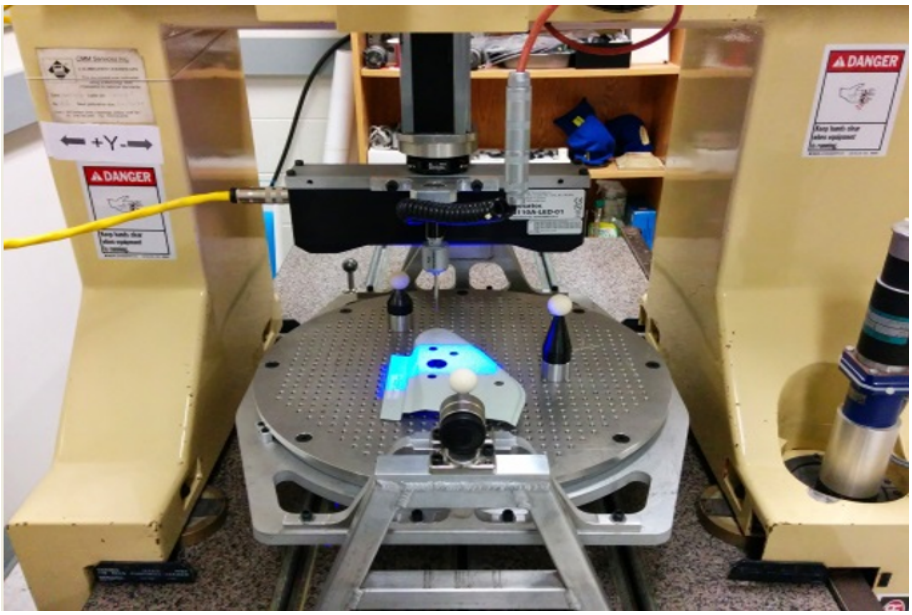


Figure 2.5: Sensor mounted on the CMM

2.4.1 Specifications

The LMI Gocator 3100 snapshot sensor, used in this work, has a near field of view of 60 mm x 105 mm and a far field of view of 160 mm x 90 mm. Maximum resolutions are of 0.09 mm, 0.15 mm and 0.035 mm along X, Y, and Z axes respectively. It has a clearance distance of 150 mm, a measurement range of 100 mm and a maximum snapshot rate of 5 Hz. The sensor is mounted on a motorized IOTA-P Cartesian CMM. Ball-bar calibration of the CMM established a very small XY squareness error of 8.073×10^{-4} rad. The static location of the sensor is obtained from the CMM motion controller through RS232/Ethernet or using a personal computer PCI counter card connected to the position scale reader heads. Along with XYZ data, the sensor provides grayscale images of the parts. Once mounted on the CMM, the snapshot sensor has to be extrinsically calibrated using an artefact. The calibration mathematics involves estimation of Homogeneous Transformation Matrices (HTMs) and feature fitting methods which will be discussed in the following sections.

2.5 Calibration Mathematics

2.5.1 Homogeneous Transformation Matrix (HTM)

Transformation between two frames is made up of rotation and translation. Consider frames A and B. The goal is to transform point, $\mathbf{p^B}(x^B, y^B, z^B)$, in frame B to point, $\mathbf{p^A}(x^A, y^A, z^A)$ in frame A. If rotation, R_B^A, a 3 $\times$ 3 matrix, and translation $T_B^A(x_B^A, y_B^A, z_B^A)$ of frame B with respect to frame A is known, then the transformation

is

$$\mathbf{p^A} = T_B^A + R_B^A \mathbf{p^B}$$
$$\begin{bmatrix} x^A \\ y^A \\ z^A \end{bmatrix} = \begin{bmatrix} x_B^A \\ y_B^A \\ z_B^A \end{bmatrix} + R_B^A \begin{bmatrix} x^B \\ y^B \\ z^B \end{bmatrix} \tag{2.1}$$

An HTM can be used to compactly represent the same transformation. HTMs are 4 × 4 matrices, commonly used in complex rigid body transformations. They store both the rotation and translation information. An HTM in this case is shown in eqn. 2.2 where $0_{1X3} = [0 \quad 0 \quad 0]$.

$$H_B^A = \begin{bmatrix} R_B^A & T_B^A \\ 0_{1X3} & 1 \end{bmatrix} \tag{2.2}$$

This HTM transforms all points in frame B to frame A. Now, eqn. 2.1 can be rewritten as

$$\begin{bmatrix} \mathbf{p^A} \\ 1 \end{bmatrix} = H_B^A \begin{bmatrix} \mathbf{p^B} \\ 1 \end{bmatrix}$$
$$\begin{bmatrix} x^A \\ y^A \\ z^A \\ 1 \end{bmatrix} = H_B^A \begin{bmatrix} x^B \\ y^B \\ z^B \\ 1 \end{bmatrix} \tag{2.3}$$

2.5.2 CMM-Sensor Calibration

The focus of this work is calibration of the snapshot sensor on the CMM. This calibration enables easy transformation of the points collected in the sensor frame to the CMM frame. Calibration artefacts should be parts that are easy to measure by both the devices. This helps establish a common coordinate system and quick transformation between the coordinate systems of the devices. While CMMs can measure most parts, snapshot sensors pose a challenge due to their relatively lower precision and noise due to surface properties of the artefact. To calibrate, rotation and translation components of the HTM have to be calculated. Translation is related to the relative positions of coordinate frames. Rotation matrices are directly related to the misalignment angles along each of the axes with respect to the calibration artefact. Rotation matrix, R, in eqn. 2.1, is the product of individual rotation matrices along X, Y and Z axes (eqn. 2.4).

$$R = R_X R_Y R_Z \tag{2.4}$$

The goal is to, first, find the transformations from the artefact to the CMM (H_A^C) and to the snapshot sensor (H_A^S). Transformation from the snapshot sensor to the CMM, H_S^C, can then be calculated as

$$H_S^C = H_A^S (H_A^C)^{-1} \tag{2.5}$$

2.5.3 Feature Fitting

Calibration methods use feature fitting algorithms to measure features like lines and planes. Total Least Squares (TLS) fitting methods [56] are widely used for this purpose. In TLS fitting, the sum of the squares of the orthogonal distances, d, of M points to a feature, S, is minimized.

$$S = \sum_{i=1}^{M} \|d_i\|^2, \tag{2.6}$$

where, for plane fitting,

$$\begin{aligned} d_i &= (s_i - p) \cdot v \\ d_i &= (s_{i,x} - p_x)v_x + (s_{i,y} - p_y)v_y + (s_{i,z} - p_z)v_z \end{aligned} \tag{2.7}$$

is the orthogonal distance of each point, s_i, from the plane. TLS plane fitting yields point, p, on the plane and its normal direction, v. Similarly, for line fitting

$$d_i = (s_m - p) \times v \tag{2.8}$$

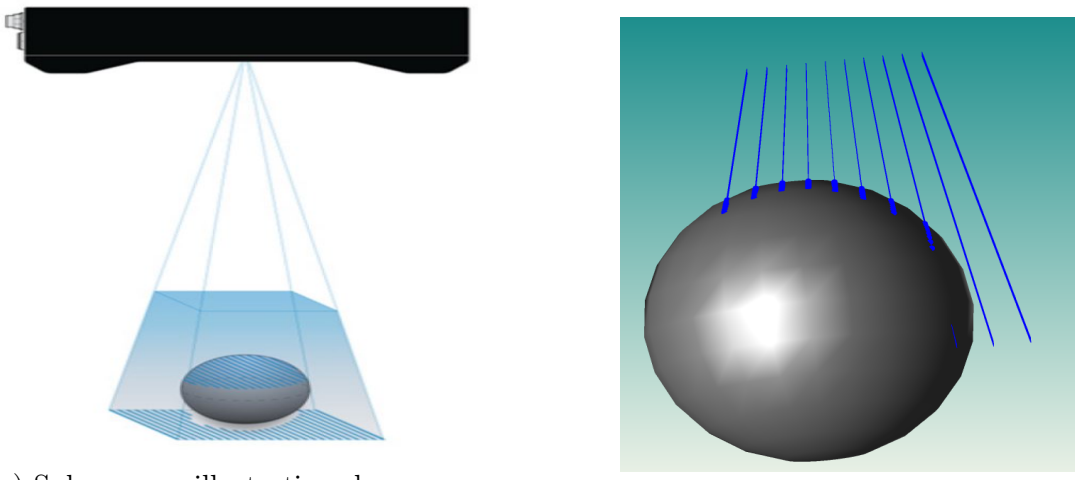
is the orthogonal distance of each point, s_i, from the line. This yields point, p, on the line and its direction, v.

2.6 Artefact Design

Several design experiments were conducted to develop a calibration artefact. This section discusses the experiments in chronological order.

2.6.1 Sphere Artefact

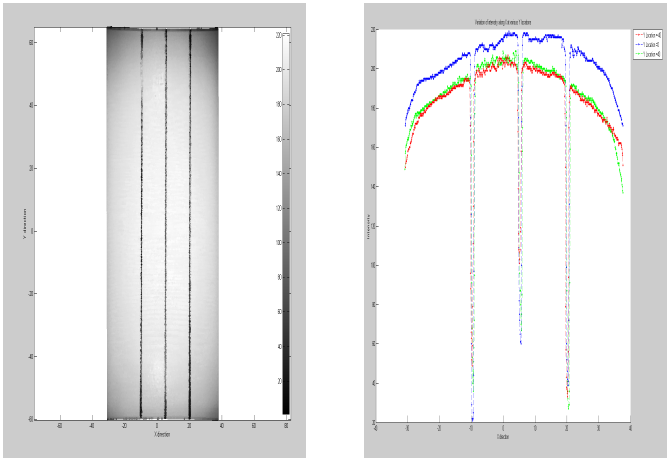
As discussed in section 2.3.4, sphere is a known calibration artefact for laser scanners. However it has been established by various studies that the surface point cloud data collected by digitizers close to the equator of a sphere is unreliable due to the steep angle of incidence (Fig. 2.6). This leads to a fit that yields a sphere that is bigger than the nominal [18, 41, 42, 63] resulting in a significant deviation in the estimation of the sphere centre. A sphere of 38.136 mm radius was scanned using the snapshot sensor. Data was fit to a sphere, using National Institute of Standards and Technology (NIST) tools, which estimated the radius to be of 37.85 mm. Hence using the sphere would not be adequate for calibration of the snapshot sensor.



(a) Sphere scan illustration shown in Gocator® documentation [21]

(b) Incident rays simulated in HOOPS®

Figure 2.6: Scanning a sphere gives unreliable points close to equator due to steep angle of incidence



(a) Gray scale image of pen lines

(b) Intensity plots at three cross sections

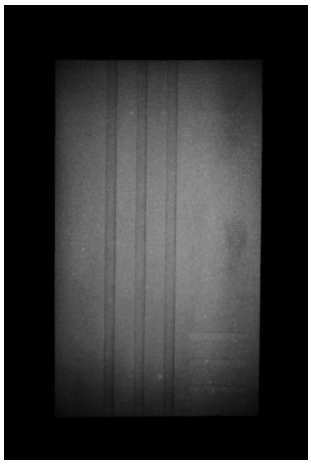
Figure 2.7: Analysis of pen lines data

2.6.2 Plane and Lines

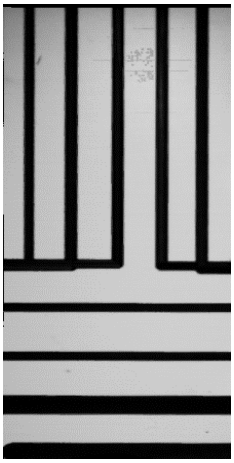
A calibration method using a combination of surface data from a planar plate and gray scale data from axially oriented lines (Fig. 2.7) was investigated. This method estimates the HTM as follows: planar surface data from the artefact is Total Least Squares (TLS) plane fit to obtain misalignments about two axes (R_X and R_Y) while misalignment about the third axis (R_Z) can be extracted by TLS line fitting gray scale line data. The preliminary experiments with this method showed promising results motivating the design and development of a calibration artifact. Further detailed experiments have revealed the challenges in gray scale implementations and an estimate of uncertainty. Research in this direction led to design of an aluminum slot calibration artefact that overcomes the drawbacks of the grayscale methods. The rest of this chapter discusses the chronological stages in the development of the working calibration artefact. Various calibration artefact designs with corresponding analytical methods, results and identified sources of uncertainty are discussed.

2.6.3 Grayscale Artefacts

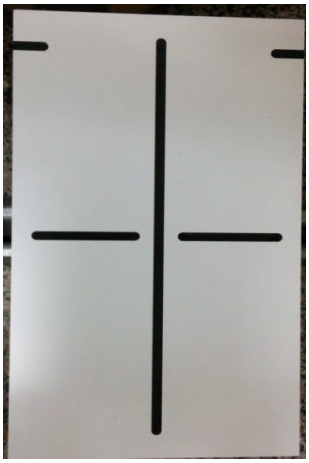
Initial investigations were performed on using ink filled slots engraved on an aluminum plate (Fig. 2.8a). Mounted on CMM table, the plate was aligned using iterative touch probe measurements. The challenges of ink filling were soon understood. While the method relies on the uniformity of ink distribution on the slots, it was difficult to maintain such uniformity with either viscous or liquid ink. However this test gave insights into design of the artefact and helped establish the alignment practices of artefacts on the CMM table.



(a) Ink filled slots on aluminium artefact



(b) ABS test part



(c) ABS artefact

Figure 2.8: Using grayscale in calibration

ABS Plate

To overcome the drawbacks of the ink filled slots, use of color layered ABS plate was suggested. ABS plastic plate used has a black layer under a white layer. Black slots can be engraved on it by machining off the top layer. Before designing the artefact, experiments were performed on a test ABS part to estimate the depth and width of

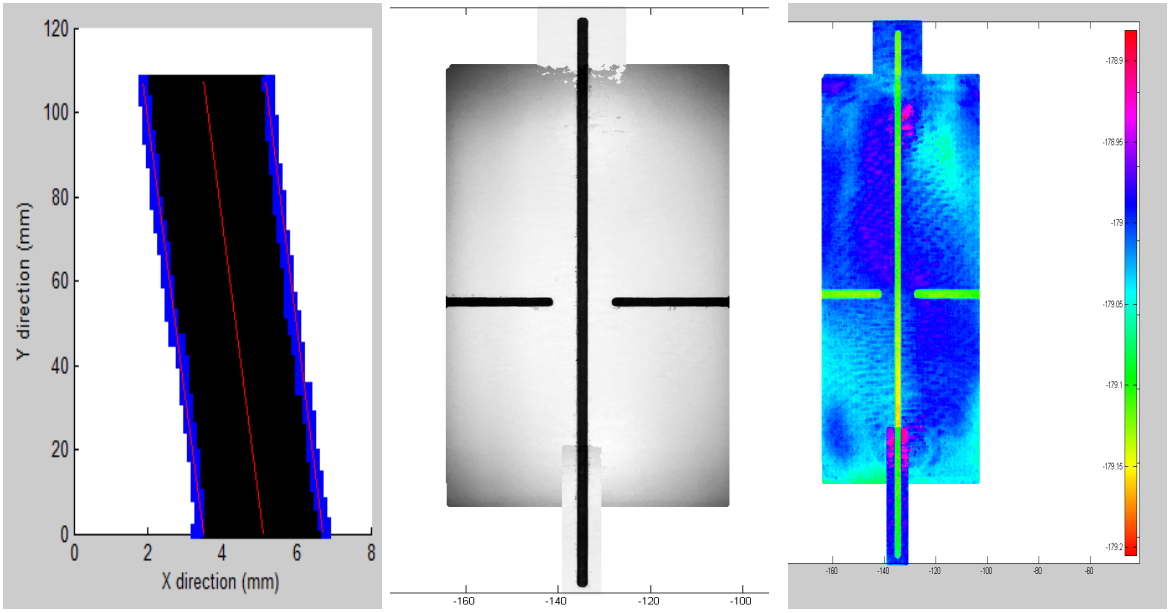
slots (Fig. 2.8b). A minimum width is essential to obtain a sufficient number of good quality pixels after removing noisy edges. The slots have to be deep enough to chip off the white layer and shallow enough to avoid shadows. Multiple lines with different depths and widths on the test part were studied. Based on the results, a calibration artefact is made with slots that are 0.1 mm deep and 3.175 mm wide (Fig. 2.8c). It is mounted on the CMM table using Renishaw AM1 [53] adjustment module (Fig. 2.4) and aligned with the CMM. Interactive algorithms were developed to extract planar and gray scale information and estimate the residual misalignments. Along with the previous line fit methods, edge detection method was also developed to detect the left and right edges of the slots (Algorithm 1).

Algorithm 1 Edge Detection

```
for each row do
  if current pixel = black then
    black pixel point set ⟵ new black pixel point
    if previous pixel = white then
      left edge point set ⟵ new left edge point
    end if
  else
    if next pixel = white then
      right edge point set ⟵ new right edge point
    end if
  end if
end for
```

It was found that the result of TLS line fit of a slot pixel is close to the mean of the values from the edge pixel line fits (Fig. 2.9a). Using ABS plate plane-grayscale method, three snapshots of the part were taken and stitched together. The height map and gray scale data of the stitched snapshots are shown below (Fig. 2.9b, 2.9c). This method was able to correct the Z direction misalignment up to 0.1 degree.

The data was analyzed to identify the sources of uncertainty. First source is



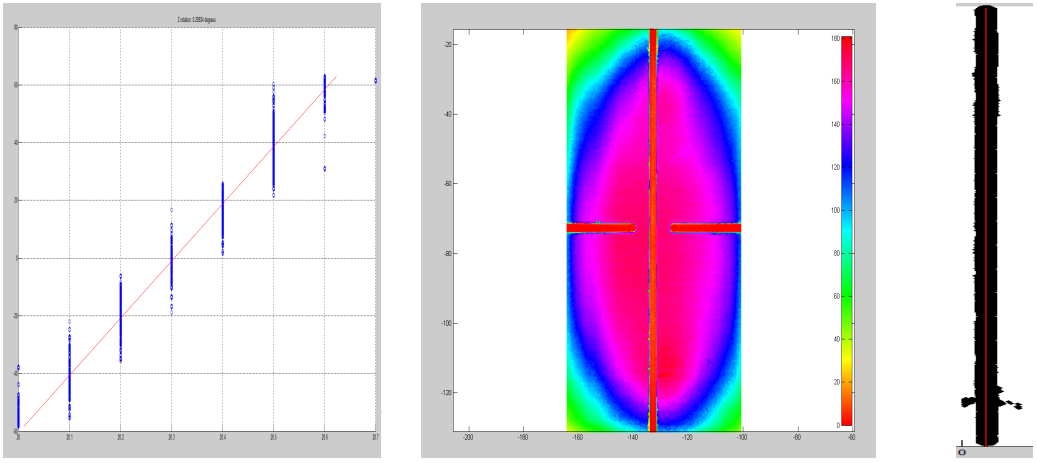
(a) Edge detection and line fits on monochrome data

(b) Registered snapshots: grayscale colormap

(c) Registered snapshots: height colormap

Figure 2.9: Analysis of ABS artefact

the flatness of the planar surface. The unevenness can be observed in the stitched snapshot (Fig. 2.9c). Although developed with high proficiency, since the artefact is made of plastic glued on metal, it is difficult to match the flatness of a machined metal part. The second source of uncertainty is the intensity distribution of the sensor in a snapshot. The intensity of an image varies radially affecting the edge detection at the ends of the slot, leading to uncertainty in line fit (Fig. 2.10b, 2.10c). The third source is the resolution of the sensor which places points in bins of 0.1 mm leading to uncertainty in the fit result (Fig. 2.10a).



(a) Edge pixels along slot length and line fit

(b) Intensity distribution in a sample snapshot

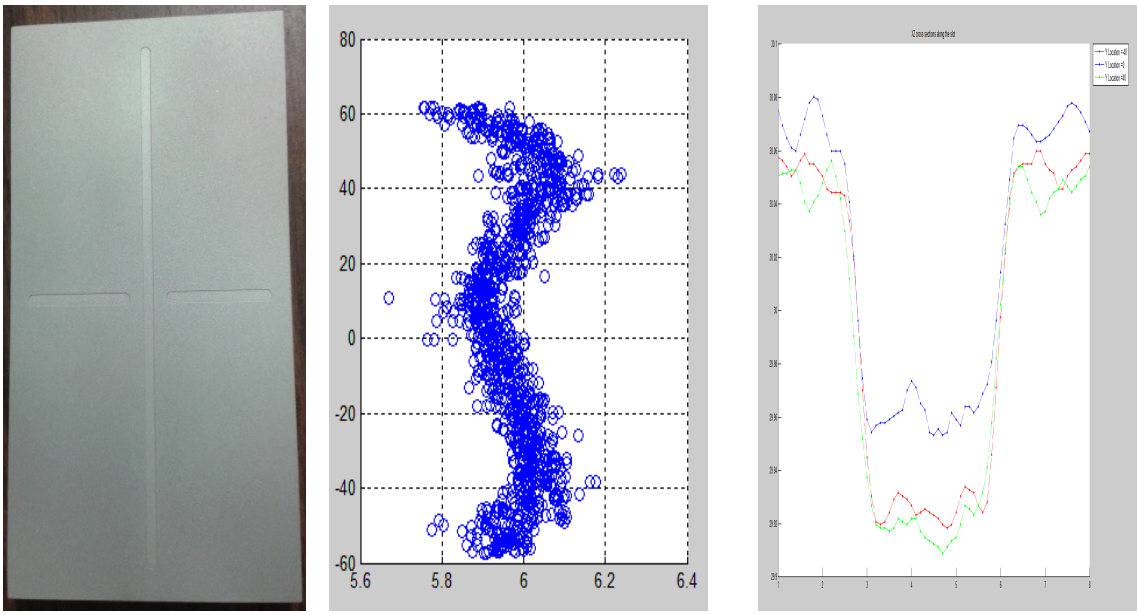
(c) Line fit of slot showing noisy ends

Figure 2.10: Uncertainty analysis of slot data from the ABS artefact

2.6.4 Aluminum Artefact

After understanding the limitations of the ABS artefact, an immediate motivation was the investigation of a similarly designed aluminum plate to overcome the drawback of uneveness (Fig. 2.11a). Interactive algorithms developed for ABS plate were adapted to calibrate the sensor using the aluminum artefact. Due to high flatness, the misalignments about the X and Y axes were corrected to less than 0.01 degrees. Sub-resolution edge detection was performed by using a combination of height based thresholding and moving window based data smoothing (Fig. 2.11c, Eqn. 2.9), with window sizes 5 and 7. The smoothed value of i^{th} pixel, $p_{i,avg}$, can be obtained by averaging it over a window size, m, as follows

$$p_{i,avg} = \frac{\sum_{i=1}^{m} p_{i-m} + \ldots + p_i + \ldots + p_{i+m}}{2m+1} \tag{2.9}$$



(a) Aluminum artefact

(b) Point distribution on a long slot edge

(c) Variation in Z values along three cross sections

Figure 2.11: Uncertainty analysis of aluminium slot data

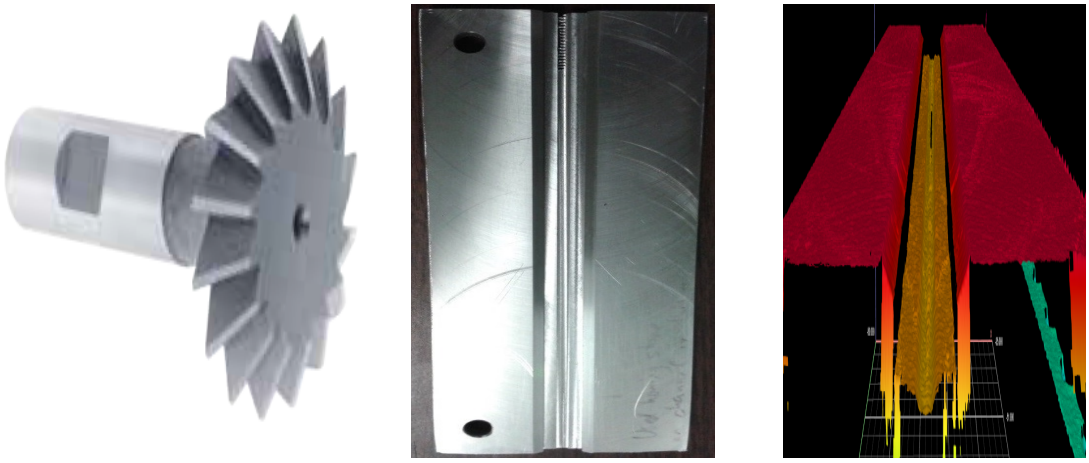
Uncertainty Analysis

Detailed study was performed on the slot edges to identify the sources of uncertainty. It was observed that the location of slot edge varied along the length of the slot (Fig. 2.11b). This is because of the non-square slot edges caused by an imperfect/curved tool-tip corner. Due to these reasons, the Z misalignment uncertainty still remained close to 0.1 degrees. To overcome the difficulty an artefact design with inclined edges was proposed.

2.6.5 Angled Slot

An angled slot is made with a double edged shank cutter with an included angle of 120 degrees (Fig. 2.12a). This slot is scanned and probed. The lines of intersection

of the inclined planes of the slot can be used to estimate the misalignment along Z direction (pointing out of the plane). Calibration using this artefact is expected to be more robust due to the density of points obtained on the inclined edges.



(a) Double edged cutter [15]

(b) Test angled slot artefact

(c) Snapshot as seen in the sensor interface

Figure 2.12: Inclined slots on test artefact

Initially, a test part is studied to understand the feasibility of the method (Fig. 2.12b, 2.12c). Methods for scanning, touch probing and analyzing the resulting data have been developed for the part. Experiments on the test part showed that the sensor is able to capture good quality point data in the regions close to the sensor origin. This data is processed in Geomagic software [20] to obtain the plane normals and the lines of intersection. Analysis of the data showed good agreement between the included angles obtained from sensor and touch probe, motivating the design of an artefact.

(a) Artefact mount (b) Sensor mount

Figure 2.13: Mechanical alignment of the calibration artefact and the sensor

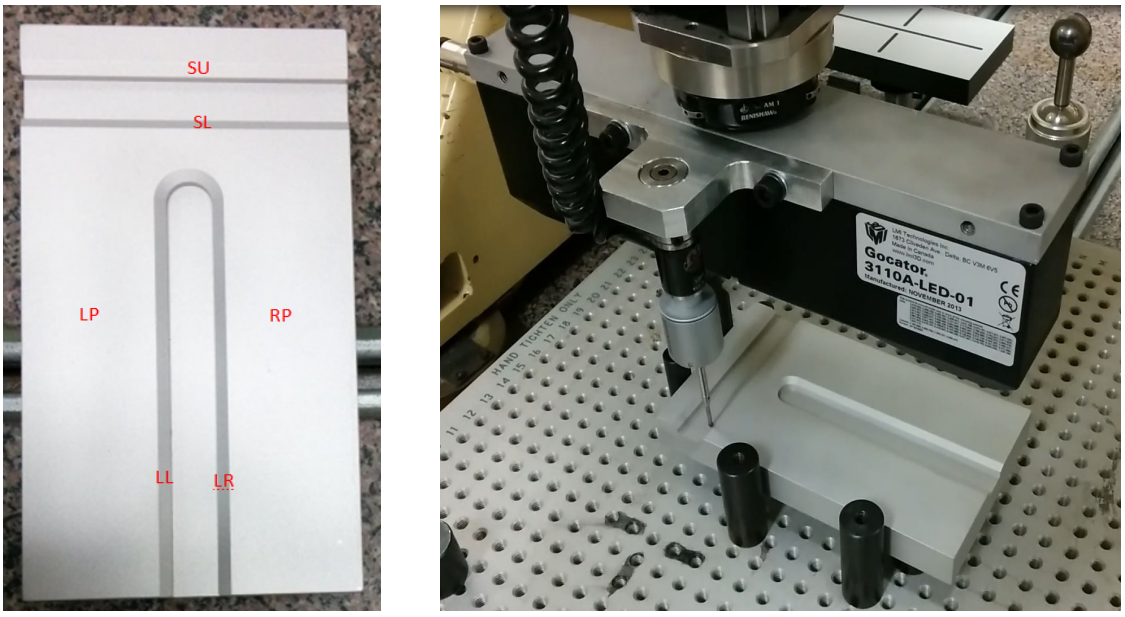
2.7 Calibration Method

2.7.1 Artefact Design

Based on the observations from the test part, an angled slot artefact has been designed and machined (Fig. 2.14). Two slots, a long slot and a short slot, with inclined edges were made using the 120 degree double edged shank cutter (Fig. 2.12a). The CAD drawing of the calibration artefact can be found in appendix B.

2.7.2 Mechanical Alignment

Before beginning the calibration process, the calibration artefact and the sensor have to be mechanically aligned to the CMM as closely as possible. In order to align the calibration artefact, it is mounted on the CMM table using an AM1 adjust module (Fig. 2.4, 2.13a). Vertical and horizontal surfaces of the artefact are touch probed to estimate the misalignments. Next yaw, pitch and roll angles of the AM1 module are adjusted to nullify the misalignments. This process is performed iteratively until the errors are too small to be corrected mechanically. Once the artefact is mechanically



(a) Artefact features (b) Touch probing the artefact

Figure 2.14: Angled slot calibration target. Details of the design can be found in the CAD drawing in appendix A

.

aligned, it is scanned using the sensor which is mounted on the CMM using another AM1 module (Fig. 2.13b). The data is studied to estimate the misalignments which are nullified using the sensor's AM1 module. This process also takes place iteratively until the errors are too small to be corrected mechanically. The residual alignment errors are compensated using the calibration process. While mechanical alignment is a one-time process, calibration has to be performed whenever the CMM is restarted.

2.7.3 Developed Protocol

The calibration procedure is described below and the related artefact labels are shown in Fig. 2.14a.

- Scan and touch probe left plane (LP), right plane (RP), left edge of long slot

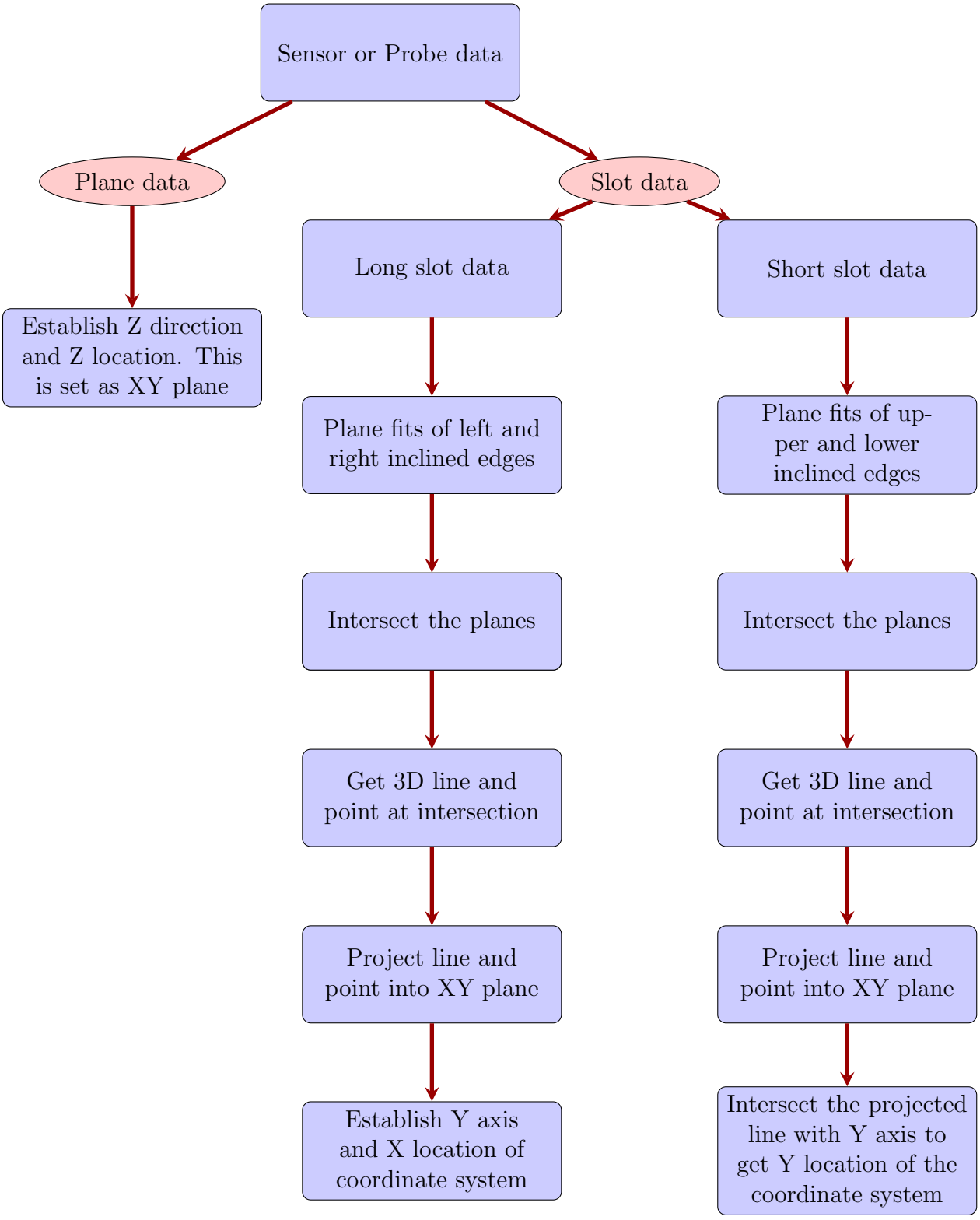


Figure 2.15: Calibration method

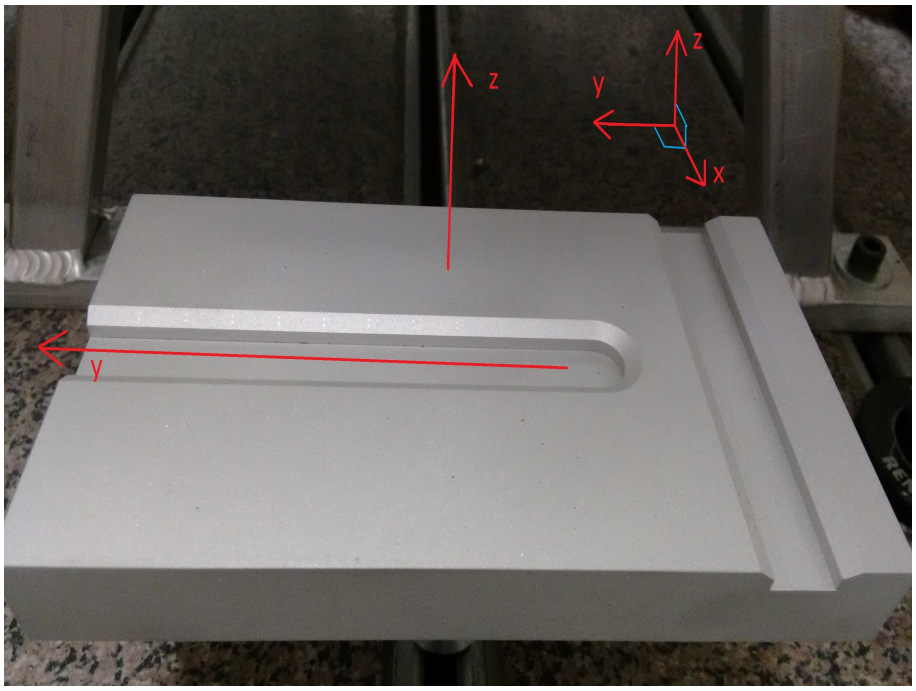


Figure 2.16: Constraction of coordinate frame on the artefact.

(LL), right edge of long slot (LR), upper edge of short slot (SU), and lower edge of short slot (SL). The following analysis is performed independently on both the scanner and the touch probe data.

- Data from flat planes (LP and RP) is TLS plane fit. The obtained normal direction and the centeroid establishes the Z direction and the Z location of the target coordinate axes. Intersection line of the LL and LR inclined planes gives Y direction and X location of target coordinate system. Intersection line of SU and SL inclined planes intersects the Y axis at the origin of the target coordinate system. X direction is assigned to be the direction orthogonal to Y and Z directions. Details of the algorithm are shown in Fig. 2.15 and Fig. 2.16.

- This establishes transformations from the sensor and the CMM coordinate systems to the calibration target. The required calibration HTM between the sensor and the CMM is obtained using the eqn 2.5.

Using this method uncertainty less than 0.01 degrees in all three directions was achieved. All the algorithms are implemented in MATLAB software [43]. The implementation code can be found in Appendix A.

2.7.4 Advantages

The calibration method has many advantages.

- From the manufacturing point of view, the calibration artefact has a simple design.
- It is easy and cheap to manufacture with the slot making tool, without the need of high skill.
- From the calibration point of view, a high density sensor point cloud data is used. Hence this method is more robust than the previous methods.
- The features on the calibration target are easy to locate with the touch probe.

The calibration method makes a CMM and snapshot sensor system with applications to sheet metal.

2.7.5 Application: Registration

The calibration transformation matrix is used to transform the local scanner data into global CMM frame. Seven snapshots of a car part, Fig. 2.17, have been captured

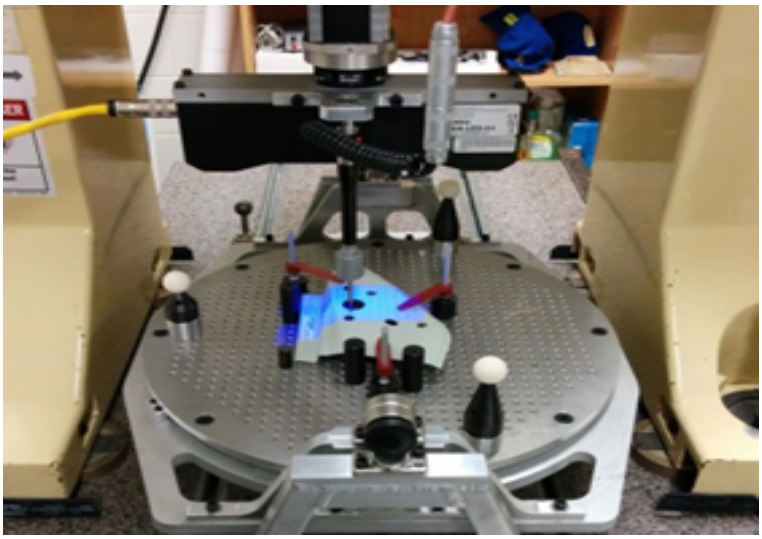


Figure 2.17: CMM-sensor system is used to scan the automotive part.

at different orientations to cover all the part. The combined data is registered into the global or CMM frame. Registered data is compared with its model in Geomagic Qualify software [20] and the results are shown in Fig. 2.18. It can be observed that most of the part deviates less than 0.380 mm from the CAD nominal model. This is good since the uncertainty per snapshot due to sensor's resolution can be between 0.1-0.16 mm. Higher deviations in the result are possibly due to deflection of the part from the CAD nominal.

2.7.6 Application: Hybrid Sensor

The second application is the hybrid sensor technique where scanner is used, before touch probing, to obtain a close initial guess of hole location on a part that significantly deviates from it CAD nominal. This saves on touch probing time and avoids probe crashes. The part is first scanned and registered to global frame using the calibration HTM. Approximate hole locations are obtained from the data. The CMM

touch probe uses this information to get close to hole location and to iteratively converge on the exact location. As tabulated results indicate (Table 2.1), when hole positions deviate up to 0.458 mm from nominal, the scanner is able to locate them within 0.107 mm of the actual position. The hole locations numbered 1-5 can be found in Fig. 2.18.

Table 2.1: Hybrid sensor application

Hole	**CAD Nominal**			**Blue LED**			**Touch Probe**		
	X	**Y**	**R**	**X**	**Y**	**R**	**X**	**Y**	**R**
1	112.012	-57.300	5.000	111.839	-56.805	5.125	111.881	-56.845	4.957
				converged			111.882	-56.842	4.957
2	101.342	-80.086	5.000	101.105	-79.688	5.126	101.131	-79.787	4.959
				converged			101.133	-79.785	4.960
3	49.128	-58.540	5.000	49.024	-58.161	5.128	48.959	-58.268	4.949
				converged			48.959	-58.268	4.950
4	76.738	-62.618	11.000	76.493	-62.195	11.202	76.547	-62.277	10.963
				converged			76.547	-62.273	10.963
5	34.714	-126.508	5.000	34.739	-126.112	5.130	34.792	-126.148	4.971
				converged			34.793	-126.148	4.971

2.8 Contributions

In summary, this work discusses integration of a compact snapshot sensor with a CMM with a novel calibration method using an artefact with simple design. The main contributions are

- Design of a calibration artefact: Experiments have been performed on various designs of gray scale and slot artefacts to find the best-suited design for the calibration requirements.
- Algorithms and software have been developed for data analysis, calibration and registration.
- A measurement protocol has been introduced for calibration.
- The calibration method has been successfully applied in multiple snapshot registration and in hybrid sensor measurements.

During the same time as this work, He *et. al.* developed a calibration artefact for a mounted custom-made fringe sensor [24]. It is polyhedron shaped and includes angled surfaces similar to our angled slot artefact. While the alignment between their angled surfaces would be difficult to maintain during machining, our design is expected to be easier to manufacture due to the use of double edge cutter.

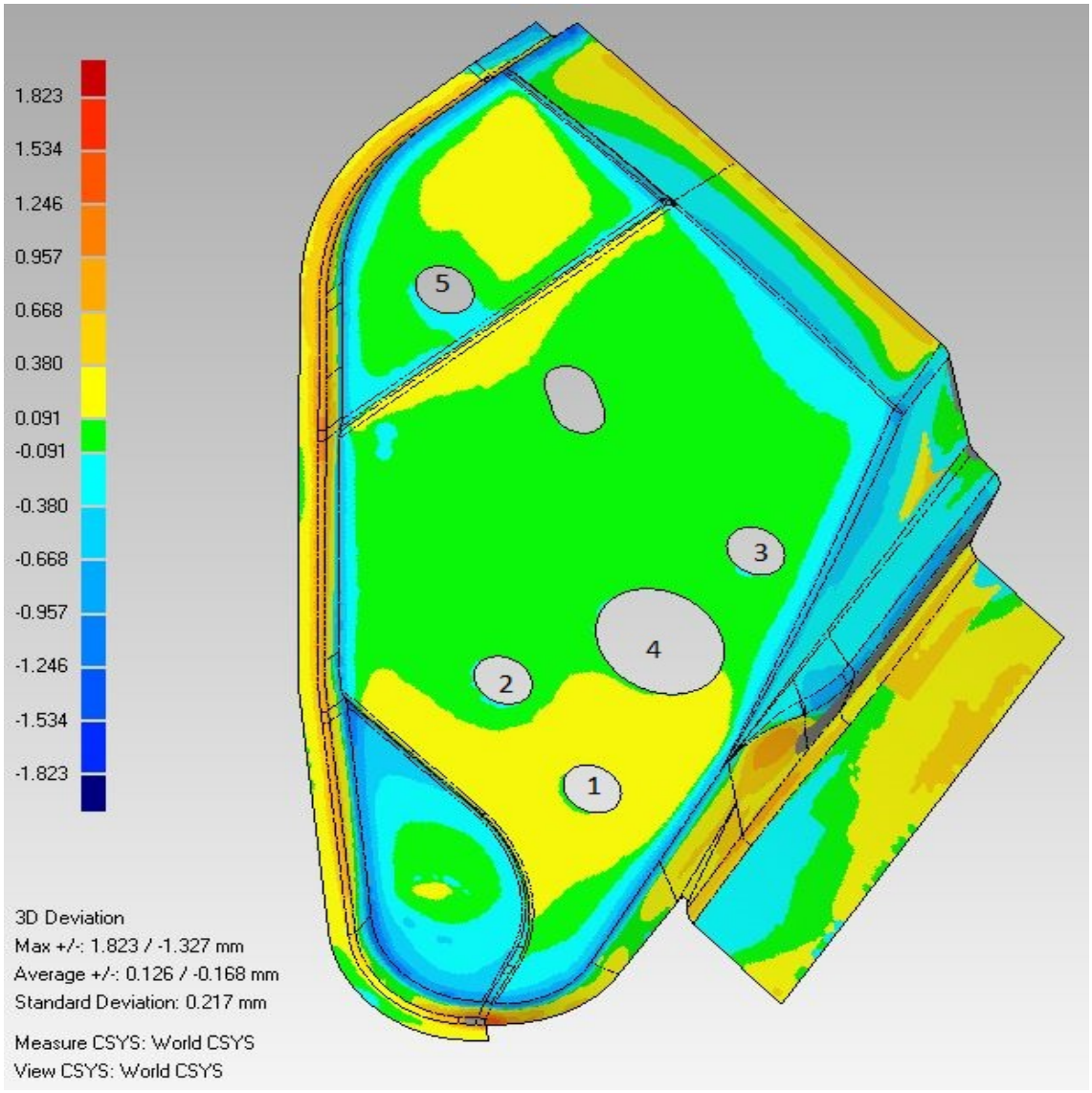


Figure 2.18: Seven snapshots of an automotive part are taken at different orientations to cover the complete surface. Data from all the snapshots has been registered to the CMM frame using the calibration method. The result is compared with its CAD model in Geomagic software.

Chapter 3

Point Cloud to CAD Deviation Mapping Using GPU Computing

3.1 Introduction

Sheet metal stamping production rates approach one part every second [14], [5]. With increasing demand that comprehensive geometric quality conformance information be approved by final assembly plant management prior to shipment, the associated digitizing analysis of millions of points requires extremely fast algorithms. For example, an industrial blue LED snapshot sensor can acquire 1 million points per second. At a 0.1 mm nominal point spacing, for even small part areas, many millions of points need to be registered with the 3D coordinate system of the CAD nominal surfaces. The memory and computing power needed to perform this analysis at part production rates far exceeds the capacity of the conventional personal microcomputer CPUs. This work investigates the alternative of using massively parallel Graphical Processing Unit (GPU) hardware. Use of this hardware exploits the parallel GPU

architecture to accelerate data intensive computations. Designed for high graphics intensity CAD and gaming, a GPU has a complex memory and processing architecture. Hence effective programming resource allocation and utilization is much more complex. Therefore, existing serial algorithms, which were not intended to run on a GPU, must be extensively rewritten. Compared to visually appealing but approximate gaming applications, dimensional metrology applications require that high accuracy be maintained throughout. Hence, while a few points may be sufficient for registration of point cloud data, all available data are helpful in part inspection.

Early GPU computing required researchers to mask arithmetic operations as graphical tasks to perform computations on CAD parts [38] or tool paths [11]. The NVIDIA CUDA programming language [55] revolutionized GPU computing. It led to applications such as rendering [23], filtering [26] and collision detection [39]. Sheet metal strain measurement was reported by Kinsner et al. [33]. Other computational applications such as distance queries between NURBS surfaces [35] and feature-fitting of geometric primitives [51], showed respectively 300X and 18X speed-up. While 2.5X - 1000X speed-up achievements are reported in literature [32], 20-30X is considered worthwhile. Erdos et al. [16] suggest GPU computing for fast mapping of CAD with point cloud data. Iterative Closest Point (ICP) like registration methods [8, 9] have already been implemented on GPUs [49]. The subsequent brute force facet-by-facet deviation estimation of the registered data is very computationally intensive. In this work, we investigated speed-up of point-facet matching (PFM) and facet-facet matching (FFM) algorithms that estimate deviations and report them as informational colormaps. These algorithms for CPU implementation are part of Origin Checkmate [13] software's proprietary library. A similar smallest sphere distance finding algorithm

showed 5X speed-up with naïve GPU implementation [30]. We show that, using a Tesla K40 GPU, careful algorithm optimization and advanced memory management delivers an impressive 124X speed-up. This work has been published in a peer reviewed journal [37].

		Point-Facet Matching	Facet-Facet Matching
Input	**CAD model**	Facets	Facets
	Actual part	Points	Facets
Output	**Deviation**	Averages of all matches	First match

Figure 3.1: Matching methods

3.2 Algorithm Challenges and Solutions

3.2.1 Algorithms

Two matching algorithms are studied here: point-facet matching (PFM) and facet-facet matching (FFM). The input and outputs of the matching methods are summarized in Fig. 3.1. PFM and FFM both accept facetted CAD model data as input. Actual part data from scanners is provided as facets and points for FFM and PFM respectively. For PFM, output is the average deviation between a model facet and its matching actual points while for FFM it is the deviation between a model facet and its first actual facet. The output is displayed as a colormap. Details of the computations that take place on every model facet in the matching algorithms are

Table 3.1: PFM and FFM matching algorithms. Here *actual* refers to actual facet and point in FFM and PFM respectively while *model* refers to a model facet. Vicinity is a fixed distance parameter.

Step	**FFM**	**PFM**
Transform	*model* to *actual*	
Binary search	Find the closest *actual* based on its distance from origin	
For all *actuals* in the vicinity of the closest *actual*		
Refined tests	Is the *actual*'s location and orientation close to that of the *model*?	
If refined tests passed	Does the *model* normal pierce the *actual*?	Estimate the projected distance between the *model* and the *actual*
Is the distance less than	Deviation threshold	*model* radius
If false	Move to the neighboring *actual* in the vicinity and go to 'refined tests' step	
Exit when	First deviation found	Average of all the deviations found

discussed in Table 3.1. The registration algorithm is based on the well-known ICP [8] method. The transformation matrix is initiated by manual matching of a few widely separated points chosen from both the digitizer and facet data. Because of the expected high number of digitizer points as compared to the size of the CAD facets, the algorithms begin by transforming the facets into the digitizer part coordinate system. This is followed by binary search of digitizer points/facets to find the actual point/-facet, j, nearest to the CAD facet. The final step is a refined search using matching parameters, in the neighborhood of j, to find matches whose deviation is within a

given threshold. The first/average of the deviations is calculated and reported as color map. As originally implemented, both the binary and neighborhood search algorithms are intricate and time consuming due to loops, branches and many memory and function calls. The algorithm complexity is $\mathcal{O}(m \log n)$ where m and n are the number of model facets and actual points/facets respectively. Actual facets/points are sorted based on their distances from origin, thus, facilitating binary search which takes $\mathcal{O}(\log n)$ time. Since the refined matches (if any) are found in close vicinity of the binary search result, the overall computational time of the algorithm per model facet remains $\mathcal{O}(\log n)$. The normal direction in point-facet matching data indicates the direction of scanning.

3.2.2 Challenges

The existing serial inspired methods face GPU challenges in both algorithm and memory implementation. Conditional tests and branches (Fig. 3.2a) exhibit poor instruction level parallelism (ILP). To address this, filtering flags were used within the GPU algorithm to bundle tests into a single branch (Fig. 3.2b). As an example, a snippet of the facet-facet matching algorithm is discussed below.

Computation 1: Z distance between CAD facet and inspection facet

Filtering Condition 1: Is the distance less than the threshold value?

Computation 2: Dot product of the model and actual facet normals

Filtering Condition 2: Dot product > 0? (normal directions within 90 degrees)

Critical Condition 1: Does the model facet normal intersect the actual facet?

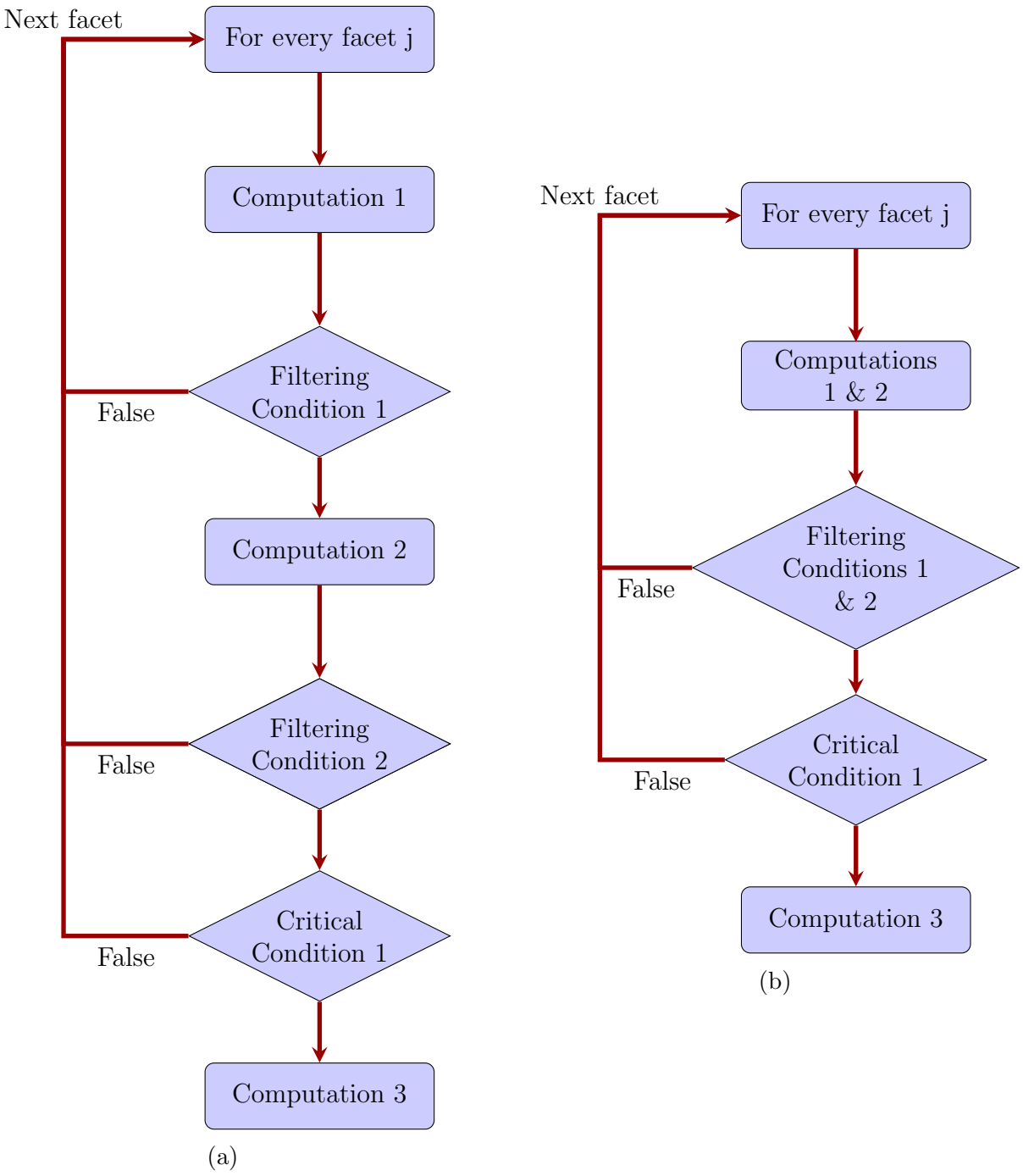


Figure 3.2: Instruction Level Parallelism (ILP) for FFM: (a) Filtering and critical condition branches in CPU implementation (b) Improving ILP on the GPU by bundling conditions

(this condition is critical as further calculations cannot proceed without the point of intersection)

Computation 3: Euclidean distance between the center of the actual facet and the point of intersection

Data structures were simplified and matching parameters were bundled for continuous memory accesses.

3.2.3 Experimental Set-up

For industrial acceptance the research described herein was implemented using an HP Z440 desktop engineering workstation, equipped with Intel Xeon E5 processor. The added NVIDIA Tesla K40 GPU card has 2880 CUDA cores. The GPU algorithms were integrated as a Dynamic Link Library (DLL) with the Origin International CheckMate software [13] added to Autodesk Mechanical Desktop, running under Microsoft Windows 7. Results from a single core of the Intel CPU were compared with Tesla K40. Each of the timing results are averaged over 100 iterations with the speed-up (s-u) calculated as shown in equation 3.1. Programming was done in Visual Studio 2013 with NVIDIA Nsight 4.1 and CUDA 6.5. Scanned part data is provided as input, and a CAD model of the part was used to generate the facets. Iterations produce colormap deviation values of every model facet. The maximum facet edge length parameter was varied to get three model data sets.

$$\text{Speed-up (s-u)} = \frac{\text{Time taken by the CPU}}{\text{Time taken by the GPU}} \tag{3.1}$$

3.3 Point-Facet Matching

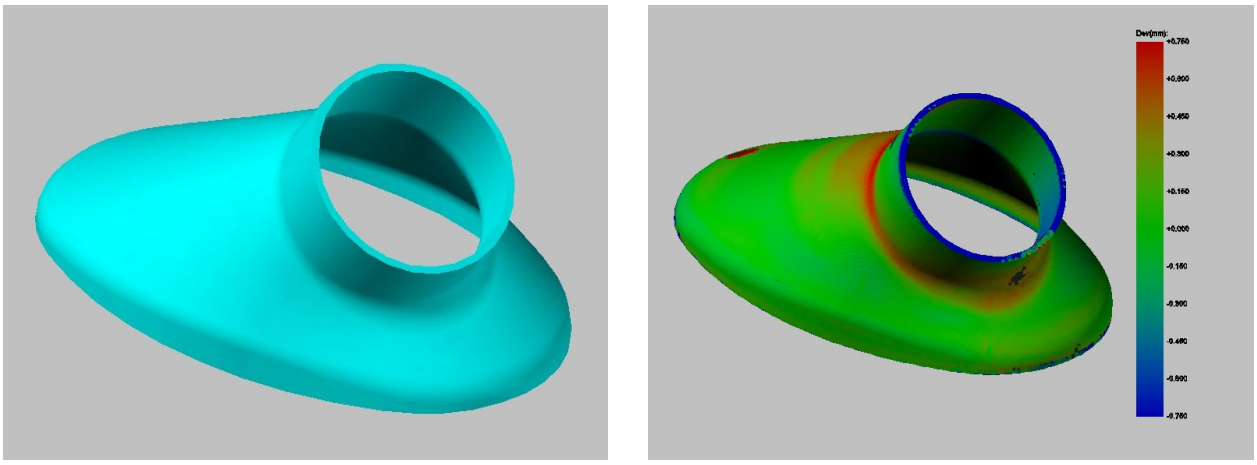
3.3.1 Initial Experiments and Results

For PFM, scanned part data from an exhaust cone with 424307 points and its CAD model (Fig. 3.3a) is provided as input. Each sample point includes the direction to the scanner to get an approximate local part orientation. Since this information is not as strong as the direction of surface normal at that point, the neighborhood search in PFM finds all possible close matches within a given threshold. The result is the average of the deviation of the model facet with each of the matched actual points (Fig. 3.3b). Three sets of test data were generated using the model facet sizes (with maximum facet edge lengths) of 2 mm, 1 mm and 0.5 mm leading to around 0.1 million, 0.3 million and 1 million model facets respectively. After improving ILP, experiments on the K40 GPU achieved speed-ups of 8X, 22X and 46X respectively. Performance analysis was conducted to understand the bottlenecks and to investigate the scope of further acceleration.

3.3.2 Performance Analysis

To understand the bottlenecks in the naïve GPU implementation, performance was analyzed using the NVIDIA Nsight tool in Visual Studio. GPU multiprocessors execute the computational instructions as sets of 32 threads called warps. On every multiprocessor, only a few of the active warps are eligible for execution. Profiling showed that, at any given instance, active warps were stalled, i.e. not eligible to move to the next instruction, due to the following reasons (Fig. 3.4):

- Memory throttle: It occurs when there are large number of pending memory



(a) Cone CAD model (b) Colormap

Figure 3.3: Point-Facet Matching: Points fit to its model to produce a colormap of deviations.

operations.

- Memory dependency: When load and store cannot happen because resources are unavailable or are being completely utilized.
- Pipe busy: It means that the load/store and arithmetic pipelines are not available for computations.
- Execution dependency: This is because input for an instruction is not available.

Hence the existing algorithm structure does not use the GPU cores effectively. The algorithm is made up of global memory accesses and arithmetic operations. A global memory access is more time consuming (200-800 clock cycles) than an arithmetic operation (1-8 clock cycles). The CUDA programming model relies on efficiently scheduling these two components to hide the latency and achieve performance. The high number of cores in the K40 can only speed-up computations. As first implemented, before a memory access is completed either the computation is already over

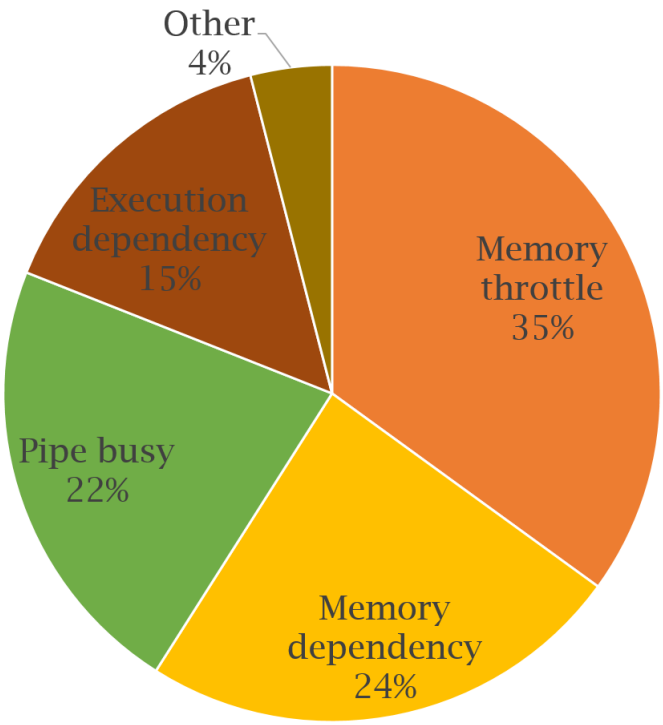


Figure 3.4: Results of performance analysis showing the warp stall reasons.

or a conditional branch is encountered. No speed-up was observed because there is not enough computation between the (many) memory accesses to hide the latency. As seen from the causes (Fig. 3.4), memory throttle and dependencies are the leading causes. The high utilization of the memory resources is keeping the pipeline busy. Common approaches to tackle these issues are bundling the memory operations, lowering memory access times, increasing ILP and allowing contiguous memory accesses. Hence to improve the performance, either the algorithm parallelism has to be increased (such as by complete looping), or the memory accesses times have to be lowered (such as using texture memory). The potential of these improvements are explored next.

3.3.3 Complete Looping

A simple way to achieve near-perfect parallelism is to eliminate branching and loop over all points, instead of a specific neighborhood (Table 3.1). This change:

- eliminates the non-contiguous memory accesses of the binary search
- removes branching in the neighborhood search

Naïve implementation of the complete looping decreased the performance (10%-70%). This is because looping over all of the points results in more total memory accesses. In attempts to overcome the drawback, use of shared and texture memories was attempted to share information of memory reads between the threads. But they could not help as the cache/memory sizes (8 KB/48 KB) of texture/shared memories are very small compared to actual point data size (∼3 MB). Due to these reasons, the complete looping approach failed to improve speed-up. Therefore, as discussed in the next section, attempts were made to lower memory access times using texture memory.

Table 3.2: Improvement in performance parameters with texture memory.

	Naïve	Texture memory
Warp occupancy (%)	5.17	20.03
Giga flops per second	26.09	99.64
Instructions per clock executed (in a GPU streaming multiprocessor)	0.14	0.66

3.3.4 Texture Memory

Overall memory access time can be lowered by re-using information. With binary and neighbourhood searches, the search range is different for each facet/thread, so shared memory cannot facilitate efficient re-use of information. Texture memory, an unconventional read-only graphics memory, is located off-chip with up to 8 MB capacity. Although located on global memory, it has an 8 KB on-chip cache making it ideal to store small, but frequently used information. Neighbouring facets share at least part of the search ranges thus exploiting texture cache. Its implementation takes additional work on the CPU side as textures support only basic data types. Texture memory produced excellent results as evident from the significant improvement in performance parameters shown in Table 3.2.

Table 3.3: PFM: Computation time (seconds) and speed-up (s-u) of Tesla K40

Number of model facets	Max. facet edge length (mm)	Intel Xeon E5 single core	NVIDIA Tesla K40			
			Naïve		Texture	
		Time	Time	s-u	Time	s-u
103,966 ($\sim$ 0.1 M)	2.000	22.6291	2.827	8	0.964	23
345,592 ($\sim$ 0.3 M)	1.000	75.1014	3.347	22	0.883	85
1,188,408 ($\sim$ 1 M)	0.500	259.126	5.676	46	2.098	124

3.3.5 Results

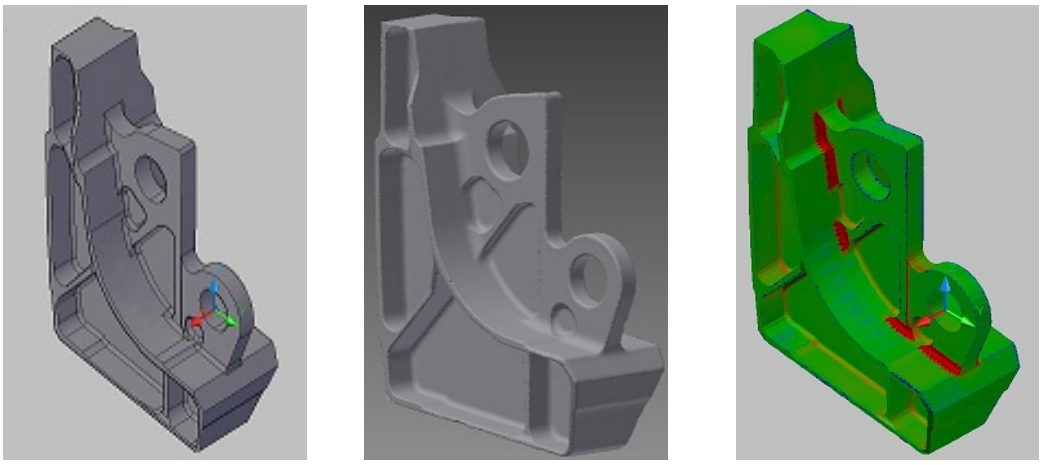
The computation times and speed-ups are summarized in Tab. 3.3. The time taken by the CPU increases with the number of points. As seen CPU takes 22 seconds, 75

seconds and around 4 minutes to process the three data sets. GPU is significantly fast. It can be seen that the naïve implementation itself brings down the computation time to 3-6 seconds. With memory optimization, computation times of 22 seconds and 75 seconds are brought under a second. Looking at the results in terms of speed up, the performance of the GPU improves with the density of the data. While naïve usage of memory, itself, showed an impressive 46X speed-up for 1 million model facets case, texture memory further boosted it to 124X. The speed-up is expected to further improve with the density of data. It can be noticed that the 0.3 M facets case takes time less than the 0.1 M case. This is likely because as the number of model facets gets close to the actual facets number ($\sim$0.4 M), fewer texture cache misses occur leading to a higher speed. A practical colormap resolution is possibly achieved when the number of CAD facets is nearly the same as the number of cloud points.

3.4 Facet-Facet Matching

3.4.1 Setup

The industrial part used for performing FFM experiments is shown in Fig. 3.5. Its actual part data contains 702429 facets each storing its normal direction. Unlike PFM where only the point and scanning direction is available, here the facet and normal information can be used to find a tighter match using various position and orientation tests. Once a match is found, the neighborhood search exits returning the deviation. This means for every model facet, FFM can find deviation sooner than PFM. While this might be computationally faster, it should be noted that such actual part facet data can only be obtained by rigorous and time consuming pre-processing of raw



(a) Model data (b) Actual data (c) FFM colormap

Figure 3.5: Input and output from facet facet matching

noisy point clouds data from scanners.

3.4.2 Results

As with PFM, the maximum facet edge length parameter of the part CAD model (Fig. 3.5a) was varied to generate three testing data sets. Preliminary experiments on FFM showed an impressive 7X, 16X and 24X accelerations for the three datasets respectively (Tab. 3.3). Learning from PFM, texture memory implementation is made without attempting complete looping. This boosted the speed-ups to 12X, 25X and 28X respectively (Tab. 3.4). For example, looking at the 1 million facets case, FFM that would take about 3 minutes on the CPU can be completed in about 6 seconds using the K40 GPU. The speed-ups are further expected to improve with the density of the data. It can be noted that the final speed-ups are lower than those observed for point-facet matching. This is because FFM algorithm has more branches and function calls. There is also at least twice as much work on the CPU side during

copying data to basic texture data types e.g. facet vertex location data is needed for FFM but not PFM. Hence PFM is more parallelizable than FFM on the GPU.

Table 3.4: FFM: Computation time (seconds) and speed-up (s-u) of Tesla K40

Number of model facets	Max. facet edge length (mm)	Intel Xeon E5 single core	NVIDIA Tesla K40			
			Naïve		Texture	
		Time	Time	s-u	Time	s-u
89,850 ($\sim$ 0.1 M)	0.157	13.596	1.876	7	1.154	12
322,010 ($\sim$ 0.3 M)	0.078	49.929	3.151	16	1.993	25
1,147,568 ($\sim$ 1 M)	0.039	174.558	7.381	24	6.343	28

3.5 Analysis and Conclusions

3.5.1 Consistency Check

Experiments were conducted to check the consistency of the GPU results with time. PFM and FFM were iterated 100 times on the GPU, resetting after each iteration. The three facet sizes were studied and iterations output deviation values of every model facet. Maximum absolute error over the iterations is estimated. Sample maximum absolute error distribution for the 1 M facet case in PFM is shown in Fig. 3.6. In all the three facet experiments, it was found to be of the order of 1E-6 for both PFM and FFM. Since the input variables are declared as single precision floats (6 significant digits), this is reasonable.

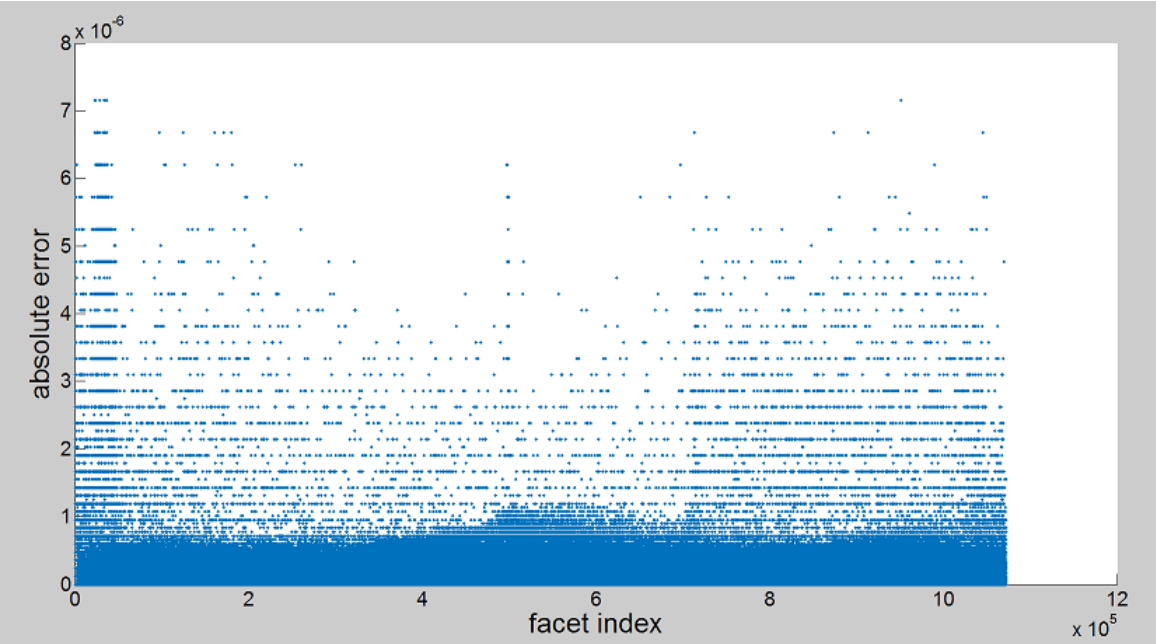


Figure 3.6: Consistency of PFM deviation results over 100 iterations for the 1 M model facet dataset.

3.5.2 CPU-GPU Result Comparison

The deviation results of PFM and FFM algorithms from the GPU and the CPU were also compared. Maximum absolute differences for each of the three test datasets are presented in Tab. 3.5. These differences are not errors and are caused by different ways of algorithm execution by the CPU and the GPU. The differences for PFM are within the limits of single floating point precision. The gap in facet-facet matching cases is relatively high likely because it uses more precision sensitive tests compared to point-facet matching and does not average deviations. Results are especially affected when the precision sensitive tests match poorly oriented actual and model facets, for example those on the corners of the geometry. Fig. 3.7 shows the absolute difference on log scale for 0.3 M dataset in FFM. Other causes for the differences could be different rounding methods and order of steps of computation executed by the CPU

and the GPU. As it can be seen from the plot, there is a drift in the differences, with values increasing with facet index. This is because the actual facets are ordered with respect to their distance from the origin. Points nearer the origin are stored with more precision than those farther from the origin resulting in the drift.

Table 3.5: Order of the maximum absolute differences of deviation results from the CPU and the GPU.

# model facets	0.1 M	0.3 M	1 M
PFM	1E-6	1E-6	1E-6
FFM	1E-3	1E-4	Two mismatches, remainder 1E-3

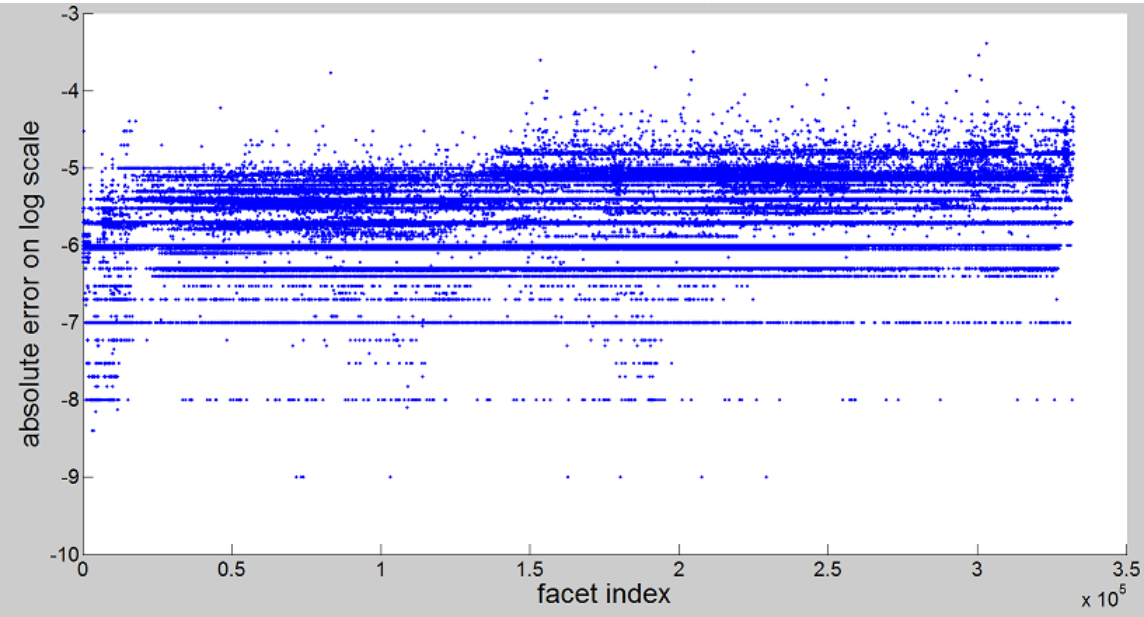


Figure 3.7: Absolute differences between the CPU and GPU deviation results for 0.3 M model facet dataset in FFM.

3.5.3 Industry Software Implementation

The research was motivated by a collaboration initiative involving Origin International Inc. [13] to improve the speed of their CheckMate measurement analysis software. The existing CPU method, initially developed for a low number of touch probe data points, was impractical for the large point clouds measured with non-contact digitizers. Accordingly, the implementation was realized using a Windows DLL that provided a bridge between the proprietary CheckMate source code and the newly developed parallel GPU code. Origin provided agreed upon function call interfaces that could switch between existing serial CPU and the newly developed parallel GPU algorithms. This approach offers a win-win university-industry collaboration.

3.5.4 Other Challenges

- Since Checkmate software is proprietary and closed, it was a challenge to create a CUDA powered Microsoft Windows DLL that can seamlessly integrate into the Origin Checkmate library which itself plugs into an Autodesk software.
- Another crucial factor to performance is the effective task distribution between the CPU and the GPU. Initially, work was divided based on the conceptual understanding and experience with the GPU programming model. Further finer distribution needed performance experiments.
- Since Checkmate software is proprietary and closed, it was tedious to track the bugs. It took longer to identify the bugs when they are located in the software instead of the DLL.
- The industry expected a minimum 8X speed-up from the GPU powered DLL

to compensate the cost overhead. Only then the product would be viable.

- Identifying the source of differences between the CPU and the GPU deviation results was another challenge (section 3.5.2). Thorough understanding of the mathematics of matching was crucial in tracking the issues.

3.5.5 Limitations

Like any work, this implementation also has a few limitations. The GPU implementation is made in CUDA programming language. While the GPU algorithms can be implemented on any GPU, the CUDA implementations run only on NVIDIA GPUs. GPUs also need high volume data for good performance. So it may not be ideal for CMM data.

3.5.6 Conclusion

In conclusion, this work demonstrates the feasibility and the method of accelerating point-facet and facet-facet matching of a dense and complex data, including colormap generation. It delivers the product as a practical and industrially applicable library compatible with Microsoft Windows.

Chapter 4

Weighted Total Least Squares

Measurement of a planar surface using conventional measuring devices like Coordinate Measuring Machines (CMMs), typically, produces uniformly sampled data which can be fitted to a plane using a simple total least squares (TLS) method. Non-contact scanners, however, typically produce point clouds with varying uniformity and uncertainty. Using a simple unweighed fit for non-uniform planar data may result in skewed or erroneous results; hence there is a need for weighted fit. This work focusses on weighted total least squares (WTLS) on parallel planes. Least squares problems on dense point clouds can take advantage of Graphical Processing Units (GPUs) for fast computation. This chapter presents an efficient GPU implementation of WTLS that processes arbitrary numbers of a) parallel planes b) sets of parallel planes and c) points per plane, making it suitable for industrial application.

4.1 Introduction

4.1.1 Fitting Methods

Three dimensional (3D) XYZ coordinate data is obtained by measuring a part using Coordinate Measuring Machines (CMMs) or digital scanners. This data is fit to various geometric shapes such as planes and spheres as well as Computer Aided Design (CAD) models to evaluate size and position tolerances. Two kinds of fitting methods, total least squares (TLS) fit and minimum zone fit, are widely studied for this purpose. Both of these methods use the concept of residual which is the orthogonal distance of a measured point to the surface of the desired geometry. TLS fit is the minimization of the sum of the squares of the residuals while minimum zone fit minimizes the value of the largest residual [2, 57]. TLS is also referred to as orthogonal least squares (OLS) or just least squares in literature. The term 'total' refers to the fact that residuals account for the distances along all the coordinate axis directions. Other methods such as particle swarm optimization, which have not been studied for dense and noisy data [61], are not considered.

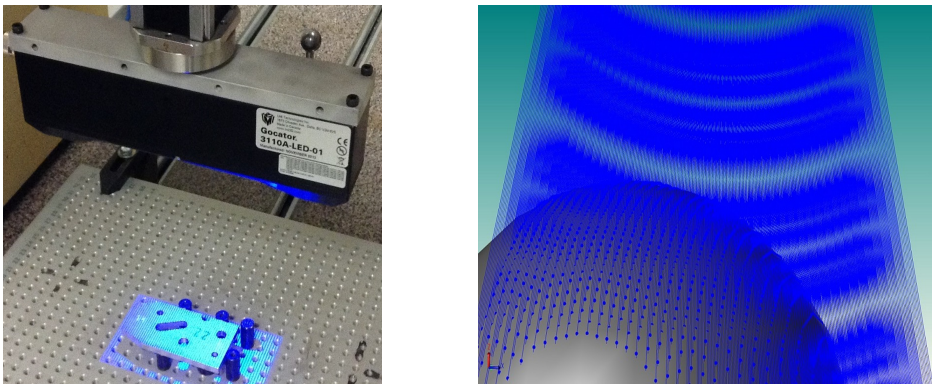
4.1.2 Minimum Zone Method

Minimum zone methods match Dimensioning & Tolerancing standards [4]. In practice, however, minimization methods have inherent drawbacks [57]. Solutions to these methods may not be unique leading to possible convergence to the wrong local minimum. Testing of such algorithms is not well developed and the implementations are computationally slow, sometimes up to a hundred times slower [60]. They are also highly sensitive to noise and outliers. In some cases, the minimum zone algorithms

may not even be available [46, 57]. Studies like [1, 40, 45] recommend minimum zone methods over least squares for accuracy. However, these works investigate only a small number of points (∼100) with no discussion of the effect of noise or of point cloud size. As a result, while this recommendation might be true for small uniform point sets without noise, it is not applicable to widely used digital devices which produce millions of noisy points. Dense data drastically slows down the computation speed of minimum zone methods. Additionally, noise increases the probability of converging to an incorrect minimum. Many of the available minimum zone methods use least squares results as initial guesses [40, 46], hence the study of least squares is valuable.

4.1.3 Total Least Squares (TLS)

Least squares techniques are widely used because they overcome most of the drawbacks of the minimum zone methods. They are less sensitive to outliers or measurement noise and are computationally fast with algorithm testing available for basic shapes, thus making them more reliable [56]. The UK's National Physical Laboratory (NPL), and the US's National Institute of Standards and Technology (NIST) have been forerunners in developing least squares methods for metrological feature fitting methods for basic geometries such as plane, line, circle, sphere, cylinder, cone and torus [19, 56]. While linear geometries have closed solutions, non-linear geometries require iterative schemes with good initial approximations [60]. Like many iteration schemes, poor initial guesses can lead to convergence to a local extremum that may not be the sought after.



(a) Blue light scanner (b) Point capture simulation

Figure 4.1: Digital sensors produce dense and non-uniform data.

4.2 Weighted Total Least Squares (WTLS)

CMMs can be programmed to sample data uniformly from a surface but they are being increasingly augmented with non-contact scanners [64]. Point sampling is typically non-uniform (Fig. 4.1). Digitizers are electro-optical devices whose accuracy of scanning is affected by part, scanner and environmental factors [31] [44]. Using simple TLS on non-uniform plane data can result in a skewed fit [58].

4.2.1 Weighting and Filtering

Consider the case of sampling and fitting two parallel planes. Uniform sampling gives the expected fit, while non-uniform sampling skews the fit in the direction of the densely sampled plane. There are two strategies to overcome this problem. One strategy is to filter the data to produce a uniform point-set, before applying TLS. Filtering can be achieved using various smoothing and data reduction techniques available in the literature [57]. Filtering leads to a uniformly distributed data set that can be studied using the simple least squares methods. The second strategy

uses careful weighing of each point followed by the application of weighted total least squares (WTLS) methods. While both could lead to the same result, WTLS can not only to address spatial non-uniformity but also uncertainties between and within data sets [58]. Moreover, NIST researchers recommend that TLS mentioned in recent and emerging ISO tolerancing standards like [28] and [29] should be interpreted as WTLS [58]; hence it is a timely opportunity to develop efficient implementations for these methods.

4.2.2 WTLS on Parallel Planes

In the literature, WTLS schemes have been studied for implicit features [3] and for uncertainty based line fitting [36]. Both the studies propose complicated iterative schemes that are developed in a general sense. Recently, researchers at NIST have developed WTLS methods with closed form solutions for parallel planes and lines, suitable for non-uniformly sampled data [58]. These methods are proven to converge to the solution that would be expected from continuous data. Uniqueness of fit is guaranteed for all reasonable data sets. This chapter focusses on their WTLS method for parallel planes which is summarized next.

4.2.3 Method

Let A and B be two parallel planes with XYZ coordinate point sets $\mathbf{x}_1, \mathbf{x}_2 \ldots, \mathbf{x}_N$ on plane A and $\mathbf{x}_{N+1}, \mathbf{x}_{N+2}, \ldots, \mathbf{x}_M$ on plane B. Here $\mathbf{x}_i$ denotes the i^{th} point (x_i, y_i, z_i). Also provided are their corresponding positive weights $w_1, w_2, \ldots, w_M$. The goal is to minimize $\sum_{i=1}^{M} w_i d_i^2$, where d_i is the orthogonal distance of $\mathbf{x}_i$ to the corresponding plane.

- Calculate the weighted centroids of both the planes as follows:

$$\mathbf{x}_A = \frac{\sum_{i=1}^{N} w_i \mathbf{x}_i}{\sum_{i=1}^{N} w_i},\ \mathbf{x}_B = \frac{\sum_{i=N+1}^{M} w_i \mathbf{x}_i}{\sum_{i=N+1}^{M} w_i}$$

- The required normal to the fit plane is the right singular vector corresponding to the smallest singular value of the $M \times 3$ matrix, $\mathbf{S}$, shown below

$$\mathbf{S} = \begin{bmatrix} \sqrt{w_1}(x_1 - x_A) & \sqrt{w_1}(y_1 - y_A) & \sqrt{w_1}(z_1 - z_A) \\ \sqrt{w_2}(x_2 - x_A) & \sqrt{w_2}(y_2 - y_A) & \sqrt{w_2}(z_2 - z_A) \\ \vdots & \vdots & \vdots \\ \sqrt{w_N}(x_N - x_A) & \sqrt{w_N}(y_N - y_A) & \sqrt{w_N}(z_N - z_A) \\ \sqrt{w_{N+1}}(x_{N+1} - x_B) & \sqrt{w_{N+1}}(y_{N+1} - y_B) & \sqrt{w_{N+1}}(z_{N+1} - z_B) \\ \vdots & \vdots & \vdots \\ \sqrt{w_{M-1}}(x_{M-1} - x_B) & \sqrt{w_{M-1}}(y_{M-1} - y_B) & \sqrt{w_{M-1}}(z_{M-1} - z_B) \\ \sqrt{w_M}(x_M - x_B) & \sqrt{w_M}(y_M - y_B) & \sqrt{w_M}(z_M - z_B) \end{bmatrix}$$

4.3 GPU Computing

In manufacturing, metrological inspection is a time-consuming process and efficient inspection methods are needed to lower production costs. Typically, digital scanning of parts produces thousands to millions of XYZ points leading to long computation times (Fig. 4.1b). Least squares operations on such dense scanner data are compute-intensive making the problem a good candidate for Graphical Processing Unit (GPU) acceleration, with memory dependent tasks performed on the CPU and compute-intensive tasks on the GPU.

4.3.1 TLS on GPU

In the literature, Mohan *et al.* implemented TLS on GPUs [51]. The authors used plane fitting methods for planes, circles, spheres, cylinders, cones and tori as presented in [56]. Performance gains of 3 to 18 times were observed for all primitives except the plane. This is due to the fact that TLS fitting of a plane is relatively less data-intensive compared to other geometries. There were some limitations to the study, such as that the data set has to be a power of two, and only uniformly distributed data was used. The GPU implementation was timed against a custom C++ implementation and not against an optimized library. No further work was conducted by the group.

4.3.2 WTLS on GPU

While performance of simple TLS on planes did not seem to improve with GPU usage in [51], it is not expected to be the case with WTLS on parallel planes due to the more data-intensive computations. In Shakarji *et al.* [58], WTLS was implemented in MATLAB without any performance oriented discussion. No GPU implementations of this method are available. The next section discusses the GPU algorithm and implementation.

4.3.3 GPU Algorithm

An implementation flow chart of the weighted parallel plane fitting algorithm is presented in Fig. 4.2. It works with an arbitrary number of planes. Singular vectors of $\mathbf{S}$, a $M \times 3$ matrix, can be calculated using Singular Value Decomposition (SVD). Mathematically, singular vectors of $\mathbf{S}$ are the same as eigenvectors of $\mathbf{S}^T\mathbf{S}$, a symmetric 3×3 matrix [62]. While SVD is tedious to perform on a GPU, matrix multiplication

to calculate $\mathbf{S}^T\mathbf{S}$ is highly parallelizable. It is therefore implemented on the GPU, followed by the power method to calculate the eigenvectors of the 3×3 matrix on the CPU.

4.4 Implementation

4.4.1 Input

Points on the planes and their weights were randomly generated. Points' X, Y and Z coordinates are random floats uniformly distributed in the ranges of (-50, 50), (-50, 50) and (-0.1, 0.1) respectively while their weights lie in the range of (1, 2). Every plane parallel to the first generated plane has Z coordinates off-set approximately by 20. Data sets of 10,000, 100,000 and 1 million XYZ points are used in the experiments.

4.4.2 Set-up

An HP Z440 desktop engineering workstation is used for this work. It is equipped with a 3.7 GHz Intel Xeon E5 processor and an NVIDIA Tesla K40 GPU card with 2880 CUDA cores. The GPU implementation (Fig. 4.2) was developed on the CUDA platform in Visual Studio 2013 with NVIDIA Nsight 4.1. The highly optimized Eigen library [22] was used in an efficient C++ implementation of the WTLS method on the CPU. This CPU implementation specifically applies the EigenSolver method in the library to calculate eigen values and vectors. Timings in milliseconds were recorded for the CPU and GPU implementations. All the reported timings are averages of those recorded over 100 iterations. The accelerations were calculated as the ratio of time taken by the CPU to that by the GPU.

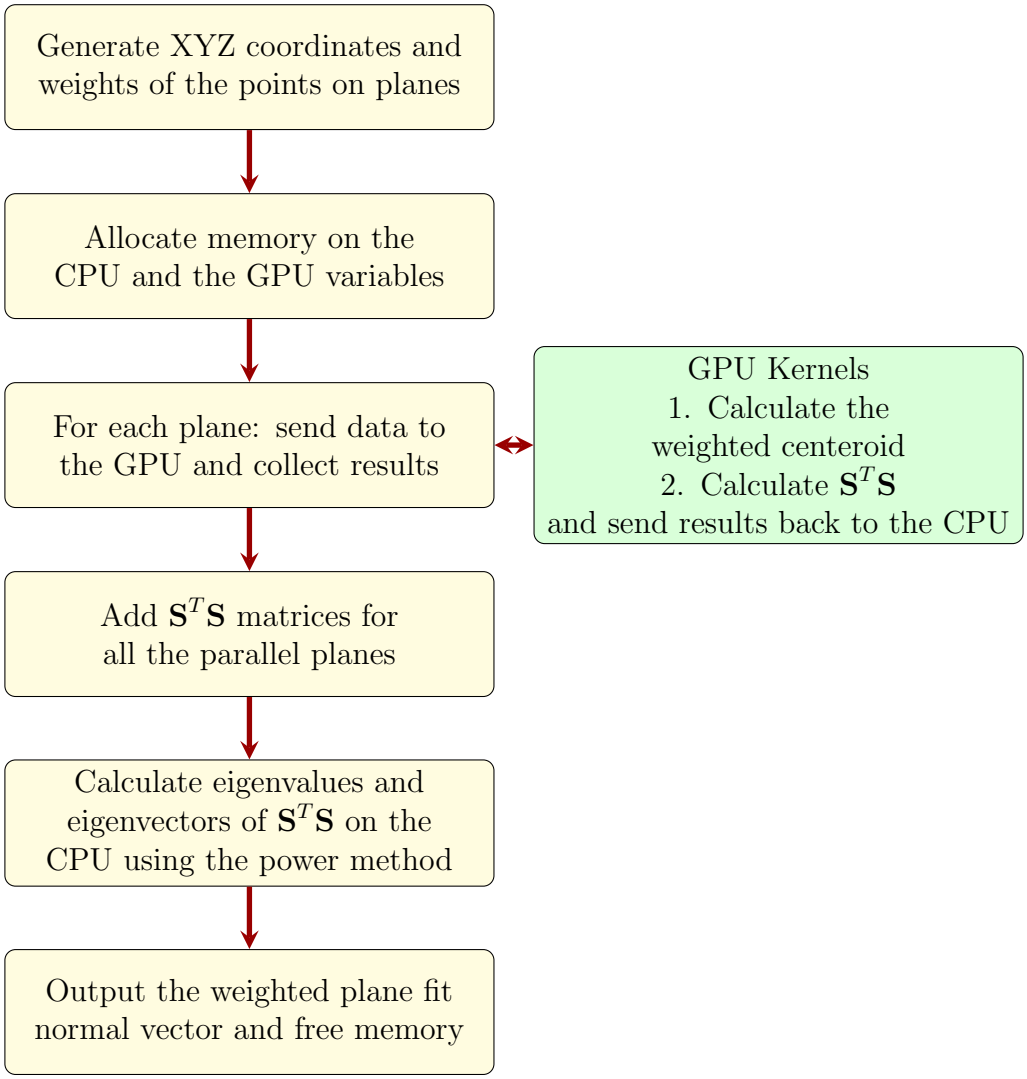


Figure 4.2: Parallel plane fit algorithm

4.4.3 GPU Implementation

The number of threads and blocks were set to be equal to the number of cores and streaming multiprocessors on the GPU respectively. The input data was approximately equally divided between the threads. Points were stored in single precision, sums and multiplications in double precision. The power method was used to calculate the required normal/eigen vector from the $\mathbf{S}^T \times \mathbf{S}$ matrices. Making it ideal for manufacturing settings, implementation can process an arbitrary number of a) plane sets b) parallel planes per set and c) points per plane. The CPU and the GPU WTLS fits were very close to each other. Normal vector differences were less than 1E-6. Since this is the maximum representable precision of the single-precision floating point input data, it means that essentially no numerical error was added by the computation.

4.5 Profiling and Optimization

This section discusses the profiling and optimization of the GPU implementation beginning with a prototype. Its performance was compared against the CPU implementation based on the highly efficient Eigen library. The incremental improvements in performance with each optimization are summarized in Fig. 4.3.

4.5.1 Managed Memory

Initially, a prototype of the implementation was developed in CUDA and tested on the GPU. Since the prototype used managed memory, it was easier to develop but not very efficient. The CPU implementation was much faster than the prototype. For

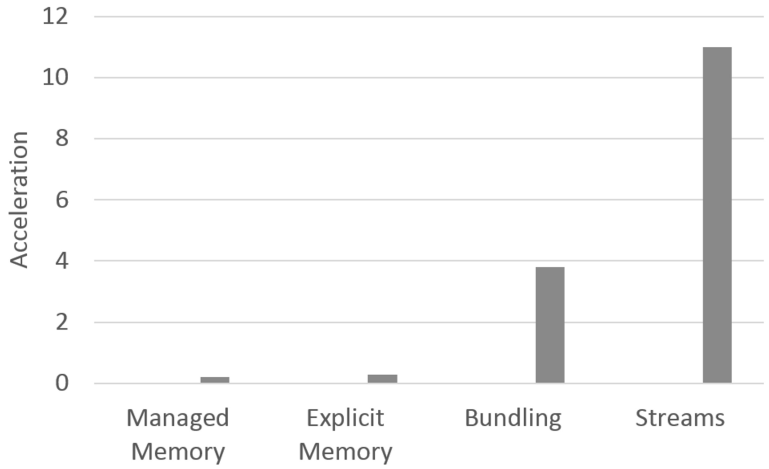


Figure 4.3: Incremental improvements in performance with each optimization

example, for one million points in each of the pair of parallel planes, the CPU was 5 times faster.

4.5.2 Explicit Memory

Refactoring the working prototype by replacing managed memory with explicit memory allocations and copies was the first step in pursuit of efficiency. This resulted in only a small improvement in speed. The CPU was still 3 times faster for 1 million points.

4.5.3 Bundling

Profiling identified memory allocations and synchronizations as bottlenecks, so input and output variables were bundled into a data structure to reduce the number of memory allocations. Many synchronizations were removed and memory copies were interleaved with CPU processes. These improvements made the GPU implementation about 4 times faster than the CPU implementation.

4.5.4 Streams

For further optimization, streams were implemented by making computations and memory copies on the two planes independent of each other. More in-depth profiling showed that memory (de)allocations account for more than 50% of the time; hence variables and storage space for one plane was reused for the next plane to save on deallocations and reallocations. With these improvements, the GPU version became 11 times faster for one million points per plane.

4.5.5 Other Optimization

Next, shared memory and atomic operations were exploited for optimization. Unlike in the last chapter, texture memory would not improve speed because the kernels were already fast (consuming less than 5% of the execution time) and the input data is accessed only once per kernel.

4.6 Results

The final implementation fits more than 2 parallel planes. Data structures, streams, memory (de)allocations and copies are made independent for each plane. In the method, parallel planes can be processed independently and sums can be accumulated at the end. They are assigned different streams for copies and computations.

4.6.1 Performance

Timing experiments with different numbers of planes showed that the profiling-directed improvements were generally beneficial. Timings were recorded for 2, 5

Table 4.1: Performance comparison. Timings in milliseconds

Points	2 planes		5 planes		10 planes	
per plane	CPU	GPU	CPU	GPU	CPU	GPU
10,000	0.9	0.2	2.7	0.3	5.3	0.5
100,000	7.3	0.9	15.6	1.7	31.3	3.2
1 million	86.8	7.7	204.9	15.8	413.8	29.7

and 10 parallel planes with 10,000, 100,000 and 1 million points each. A maximum of 14 times acceleration was observed for 1 million points. The accelerations increase both with the number of parallel planes and the number of points per plane. It is very significant performance considering that the GPU implementation is compared against a highly optimized CPU library. The detailed timings and accelerations are presented in Table 4.1 and Fig. 4.4 respectively.

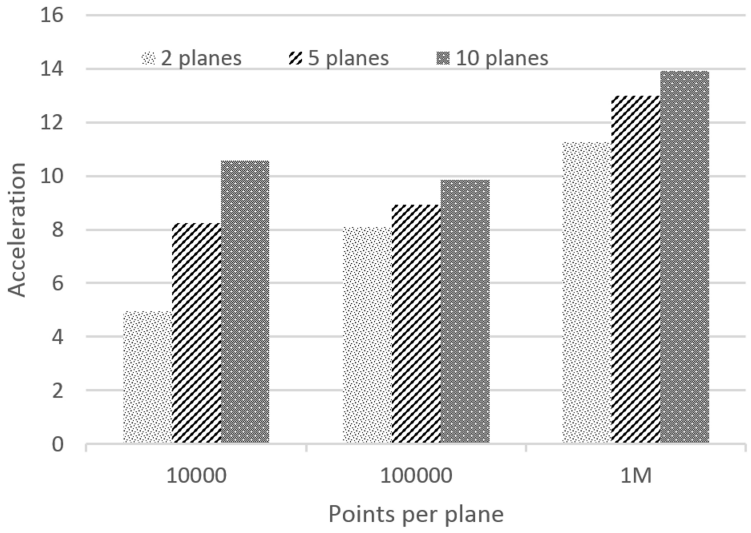


Figure 4.4: GPU acceleration compared to the CPU implementation

The results show pure CPU timing. In an industrial setting, the CPU could also be busy doing other tasks such as data acquisition and displaying a graphical interface. In this setting, the acceleration would likely be higher.

4.6.2 Limitations

The development of GPU implementations is more time-consuming than CPU implementations due to the difficulty in debugging distributed processes. While this algorithm can be implemented on any GPU, the CUDA implementation runs only on NVIDIA GPUs. Finally, GPU implementations typically benefit from dense data and will not show benefits for small data sets.

4.6.3 Future Architecture

Tesla K40 GPU came into the market more than 3 year ago and more recent processors are faster. Newer Tesla architecture is more than twice as fast (10.6 TFLOPS) as the K40 (4.29 TFLOPS), and recompiling for this processor would be expected to double the performance.

4.6.4 Applications

As discussed in beginning of the chapter, WTLS should be used to comply with the recent standards. This implementation could be used on planar datasets obtained at different resolutions and with different uncertainties. For data with different resolutions, weights could be partitioned according to the areas measured. Another major application is with large parts. When large parts are scanned in small sections, the multiple sets of point cloud data should be stitched together. In such cases, the GPU

implementation helps fit the data quickly in parallel. Fitting the sections independently helps gauge the uniformity and alignment of the planes. Certain regions can be also be avoided using zero or low weights, such as bumps, holes, corners or those with specular reflection.

4.7 Conclusions

This chapter demonstrates that GPU-accelerated WTLS for parallel planes can achieve very significant 14 times acceleration, even above the best available CPU algorithms. It also discusses the merits of least squares fitting over the minimum zone method for noisy dense data from digitizers.

Chapter 5

Conclusions

Industries like automotive and aerospace require tight tolerances on manufactured parts. Conventional measurement devices like CMM are very precise but slow. Chapter 2 discusses a calibration method to integrate a snapshot sensor on a CMM to facilitate quick inspection. A novel calibration method is developed using an artefact with simple design. Algorithms and software were developed along with a measurement protocol. Applicability of the method to industrial problems has been demonstrated.

Chapter 3 discusses a solution to an industrial problem. A point cloud to CAD comparison colormap tool is investigated to identify the feasibility of acceleration. Algorithm bottlenecks were identified and faster GPU implementations ware developed. Performance analysis and optimization was conducted to guide further improvement. The final tool was 124 times faster. The product was delivered as a practical and industrially applicable library.

Finally in chapter 4, a GPU-accelerated implementation of weighted total least squares (WTLS) for parallel planes is discussed. Initially it was implemented for two planes. Later it was extended to tackle arbitrary numbers of parallel planes,

sets of parallel planes and points per plane. After rigorous performance analysis, optimization gave incremental performance improvements. The resulting product is 14 times faster than a highly optimized C library. Finally, fields of industrial applicability were identified.

In conclusion, this work discusses accelerating two aspects of metrological inspection: sensor integration and computation. While the discussed sensor integration saves time, it is yet to be fully automated. Data filtering and noise removal from the calibration artefact scan is manual but could be automated and processed in real-time using GPU computations.

Appendix A

Calibration Algorithms

This appendix covers MATLAB [43] implementation of major algorithms discussed in Chapter 1.

A.1 Projection of Point on a Plane

```
function ProjectedPoint = ProjectPointonPlane(Point, PlaneNormal,
    PointonPlane)
%This function projects a given point on specified plane along its
% normal. Line connecting the Point (P) and its projection
% ProjectedPoint, ProjP) will be parallel to the Plane normal,N.
% Hence ProjP = P + t * N for a number 't' to be evaluated
% Equation of plane: N(X-PP) = 0; PP is given Point on Plane
% Since ProjP lies on the plane, N.(ProjP-PP)= 0
% N.(P + t * N - PP) = 0 => N.(P - PP) + t = 0 => t = N.(PP-P)
```

```
t = dot(PlaneNormal, PointonPlane)-dot(PlaneNormal, Point);
ProjectedPoint = Point + t * PlaneNormal;
```

A.2 Point of Intersection of Two Coplanar Lines (3D)

```
function OutputPoint = PointofIntersectionofCoplanarLines3D(Dir1, Point1,
    Dir2, Point2, PlaneNormal)
%This function calculates points of intersection of two given coplanar
    lines in 3D. Input: 3D Line directions and points on the coplanar
    lines; normal to the plane Normal direction is optional and can be
    calculated using cross product of directions
P0 = Point1;
Q0 = Point2;
PUnitVec = Dir1/norm(Dir1);
QUnitVec = Dir2/norm(Dir2);
%Any point on P line can be written as P0 + t *PUnitVec, t is a real number
%Any point on Q line can be written as Q0 + s *QUnitVec, s is a real number
% Let IntersecPQ be the point of intersection of lines
% Let QUnitVecPerp be a coplanar unit vector perpendiclar to QUnitVec
QUnitVecPerp = cross(PlaneNormal, QUnitVec);
%Since IntersecPQ lies on Line Q, dot((IntersecPQ-Q0), QUnitVecPerp) = 0;
% => dot((P0 + t *PUnitVec -Q0), QUnitVecPerp) = 0 since IntersecPQ also
    lies on Line P
```

```
% => dot((P0 - Q0), QUnitVecPerp) + t * dot(PUnitVec, QUnitVecPerp) = 0
t = (dot((Q0 - P0), QUnitVecPerp))/(dot(PUnitVec, QUnitVecPerp));
IntersecPQ = P0 + t *PUnitVec;
OutputPoint = IntersecPQ;
```

A.3 Point on Line of Intersection of Two Planes

```
function OutputPoint = PointonLineofIntersection(Point1, Normal1, Point2,
    Normal2)
%This function calulates the point on Line of intersction of two given
%planes. X is the unknown point on line of intersection. P = [Point1 ;
    Point2]
%Equation of planes satisfy N(X-P) = 0; so we need to solve NX = NP
N = [Normal1; Normal2];
NP = [dot(Normal1, Point1) ; dot(Normal2, Point2)];
OutputPoint = N\NP;%X
OutputPoint = OutputPoint'; %outputting in same format as input
end
```

A.4 Select Data in a Point Cloud Plot

```
function BrushedData = CopyBrushedData(gobj)

%Returns brushed data from the figure object gobj;
```

```
%adapted and simplified from datamanager.copyUnlinked found in
%C:\Program Files\MATLAB\R2014b\toolbox\matlab\datamanager\@datamanager
%Last Revision date 14 Nov 2014

import com.mathworks.page.datamgr.brushing.*;

% Find brushed graphics in this container
sibs = datamanager.getAllBrushedObjects(gobj);
BrushedData = [];

for i = 1: length(sibs) %loop over the brushed objects
    gobjBrushed = sibs(i);
    if ~graphicsversion(gobjBrushed,'handlegraphics')%If figures return
        objects; ie if matlab version >= R2014b
        BrushedDataSection = brushing.select.getArraySelection(gobjBrushed);
    else
        this = getappdata(double(gobjBrushed),'Brushing__');%If figures
            return handles; ie if matlab version <= R2014a
        BrushedDataSection = this.getArraySelection;
    end
    BrushedData = [BrushedData; BrushedDataSection];
end
```

A.5 Estimation of Calibration HTM using Processed Data from Sensor and CMM

```
%This code is use to obtain the calibration HTM using calibration artefact
    data from sensor and CMM
%Sensor Data
%Data in sensor frame; sensor uses LHS so left and right are
%opposite compared to sensor
%Plane equation (SenPointonPlane - (x,y,z)).SenPlaneNormal = 0;
%Line equation P = P0 + t.Pvec
%HTM from transforming from sensor frame to part frame (origin located at
    SenProjectedSlotIntersectionPoint)
%HTM = [R T]
%      [0 1]
%R = [(cross(PlaneNormal,LongSlotUnitVect))' ProjectedSenLongSlotUnitVect'
    SenPlaneNormal']
% T = SenProjectedSlotIntersectionPoint'
%CMM data; data in RHS frame
%Touch probe data processes and Output data from DataProcess_v4 included as
%InputForCalibrationHTMEstimation.m
%Similarly sensor input from SensorInputForCalibrationHTMEstimation.m


clear all
CurrentFolder = uigetdir('C:\Users\kurellv\Dropbox
    (McMasterDMSL)\TeamSharedFolder\Kai\DailyPlaneCalibration\');
cd(CurrentFolder)
```

```
format longG
diary('HTMlog.txt');
fprintf('%s \n', datestr(now)); %display date and time

%sensor input provided from local SensorInputForCalibrationHTMEstimation.m
%file
SensorInputForCalibrationHTMEstimation

%Finding Long slot line equation (Point and Direction):
SenLongSlotDir = cross(SenLeftLongSlotNormal,SenRightLongSlotNormal);%may
    not be a unit vector
SenLongSlotPoint = PointonLineofIntersection(SenLeftLongSlotPoint,
    SenLeftLongSlotNormal,SenRightLongSlotPoint,SenRightLongSlotNormal);

%Finding Short slot line equation (Point and Direction):
SenShortSlotDir =
    cross(SenAboveShortSlotNormal,SenBelowShortSlotNormal);%may not be a
    unit vector
SenShortSlotPoint = PointonLineofIntersection(SenAboveShortSlotPoint,
    SenAboveShortSlotNormal,SenBelowShortSlotPoint,SenBelowShortSlotNormal);

%Directions and points projected onto planes; directions may not be unit
    vectors
ProjectedSenLongSlotDir = SenLongSlotDir -
    dot(SenLongSlotDir,SenPlaneNormal)*SenPlaneNormal;
```

```
ProjectedSenShortSlotDir = SenShortSlotDir -
    dot(SenShortSlotDir,SenPlaneNormal)*SenPlaneNormal;

ProjectedSenLongSlotPoint = ProjectPointonPlane(SenLongSlotPoint,
    SenPlaneNormal, SenPointonPlane);
ProjectedSenShortSlotPoint = ProjectPointonPlane(SenShortSlotPoint,
    SenPlaneNormal, SenPointonPlane);

%Point of intersection of projected lines
SenProjectedSlotIntersectionPoint =
    PointofIntersectionofCoplanarLines3D(ProjectedSenLongSlotDir,
    ProjectedSenLongSlotPoint, ProjectedSenShortSlotDir,
    ProjectedSenShortSlotPoint, SenPlaneNormal)

ProjectedSenLongSlotUnitVect =
    ProjectedSenLongSlotDir/norm(ProjectedSenLongSlotDir);
SenR = [ cross(ProjectedSenLongSlotUnitVect', SenPlaneNormal')
    ProjectedSenLongSlotUnitVect' SenPlaneNormal'];
SenT = SenProjectedSlotIntersectionPoint';

SenHTM = eye(4); SenHTM(1:3,1:3) = SenR; SenHTM(1:3,4) = SenT;
SenHTM

%CMM part: Input data
InputForCalibrationHTMEstimation
```

```
%Finding Long slot line equation (Point and Direction):
cmmLongSlotDir = cross(cmmLeftLongSlotNormal,cmmRightLongSlotNormal);%may
    not be a unit vector
cmmLongSlotPoint = PointonLineofIntersection(cmmLeftLongSlotPoint,
    cmmLeftLongSlotNormal,cmmRightLongSlotPoint,cmmRightLongSlotNormal);

%Finding Short slot line equation (Point and Direction):
cmmShortSlotDir =
    cross(cmmAboveShortSlotNormal,cmmBelowShortSlotNormal);%may not be a
    unit vector
cmmShortSlotPoint = PointonLineofIntersection(cmmAboveShortSlotPoint,
    cmmAboveShortSlotNormal,cmmBelowShortSlotPoint,cmmBelowShortSlotNormal);

%Directions and points projected onto planes; directions may not be unit
    vectors
ProjectedcmmLongSlotDir = cmmLongSlotDir -
    dot(cmmLongSlotDir,cmmPlaneNormal)*cmmPlaneNormal;
ProjectedcmmShortSlotDir = cmmShortSlotDir -
    dot(cmmShortSlotDir,cmmPlaneNormal)*cmmPlaneNormal;

ProjectedcmmLongSlotPoint = ProjectPointonPlane(cmmLongSlotPoint,
    cmmPlaneNormal, cmmPointonPlane);
ProjectedcmmShortSlotPoint = ProjectPointonPlane(cmmShortSlotPoint,
    cmmPlaneNormal, cmmPointonPlane);

%Point of intersection of projected lines
```

```
cmmProjectedSlotIntersectionPoint =
    PointofIntersectionofCoplanarLines3D(ProjectedcmmLongSlotDir,
    ProjectedcmmLongSlotPoint, ProjectedcmmShortSlotDir,
    ProjectedcmmShortSlotPoint, cmmPlaneNormal)

ProjectedcmmLongSlotUnitVect =
    ProjectedcmmLongSlotDir/norm(ProjectedcmmLongSlotDir);
cmmR = [ cross(ProjectedcmmLongSlotUnitVect', cmmPlaneNormal')
    ProjectedcmmLongSlotUnitVect' cmmPlaneNormal'];
cmmT = cmmProjectedSlotIntersectionPoint';

cmmHTM = eye(4); cmmHTM(1:3,1:3) = cmmR; cmmHTM(1:3,4) = cmmT;
cmmHTM

%Converting the HTM to LHS for consistncy
 LHSProjectedcmmLongSlotUnitVect = ProjectedcmmLongSlotUnitVect;
 LHSProjectedcmmLongSlotUnitVect(1) = -LHSProjectedcmmLongSlotUnitVect(1);
 LHScmmPlaneNormal = cmmPlaneNormal;
 LHScmmPlaneNormal(1) = -LHScmmPlaneNormal(1);
 LHScmmProjectedSlotIntersectionPoint = cmmProjectedSlotIntersectionPoint;
 LHScmmProjectedSlotIntersectionPoint(1) =
     -LHScmmProjectedSlotIntersectionPoint(1);
 LHScmmR = [ cross(LHSProjectedcmmLongSlotUnitVect', LHScmmPlaneNormal')
     LHSProjectedcmmLongSlotUnitVect' LHScmmPlaneNormal'];
%
 LHScmmT = LHScmmProjectedSlotIntersectionPoint';
```

```
 LHScmmHTM = eye(4); LHScmmHTM(1:3,1:3) = LHScmmR; LHScmmHTM(1:3,4) =
     LHScmmT;
LHScmmHTM
%Visually verify that LHScmmHTM and SenHTM have similar eliments in the
%rotation part
%Concept: There are two ways to transform coordinates into calibration
%artefact coordinate system
% 1. Multiplying/transforming LHS CMM data with LHScmmHTM
% 2. Multiplying/transforming (LHS CMM location offsetted) sensor data with
% CMMtoSensorHTM * SenHTM;
%Hence CMMtoSensorHTM * SenHTM = LHScmmHTM
% Using this concept we can obtain CMMtoSensorHTM as
% CMMtoSensorHTM = LHScmmHTM * inverse(SenHTM) or using matlab function
% CMMtoSensorHTM = mrdivide(LHScmmHTM, SenHTM)
%Verify that Rotation part of CMMtoSensorHTM is close to identity upto
    four decimals
 CMMtoSensorHTM = mrdivide(LHScmmHTM, SenHTM)
fprintf('\n');
diary off
```

Appendix B

Calibration Artefact Drawing

Design of the angled slot calibration artefact is shown in the next page.

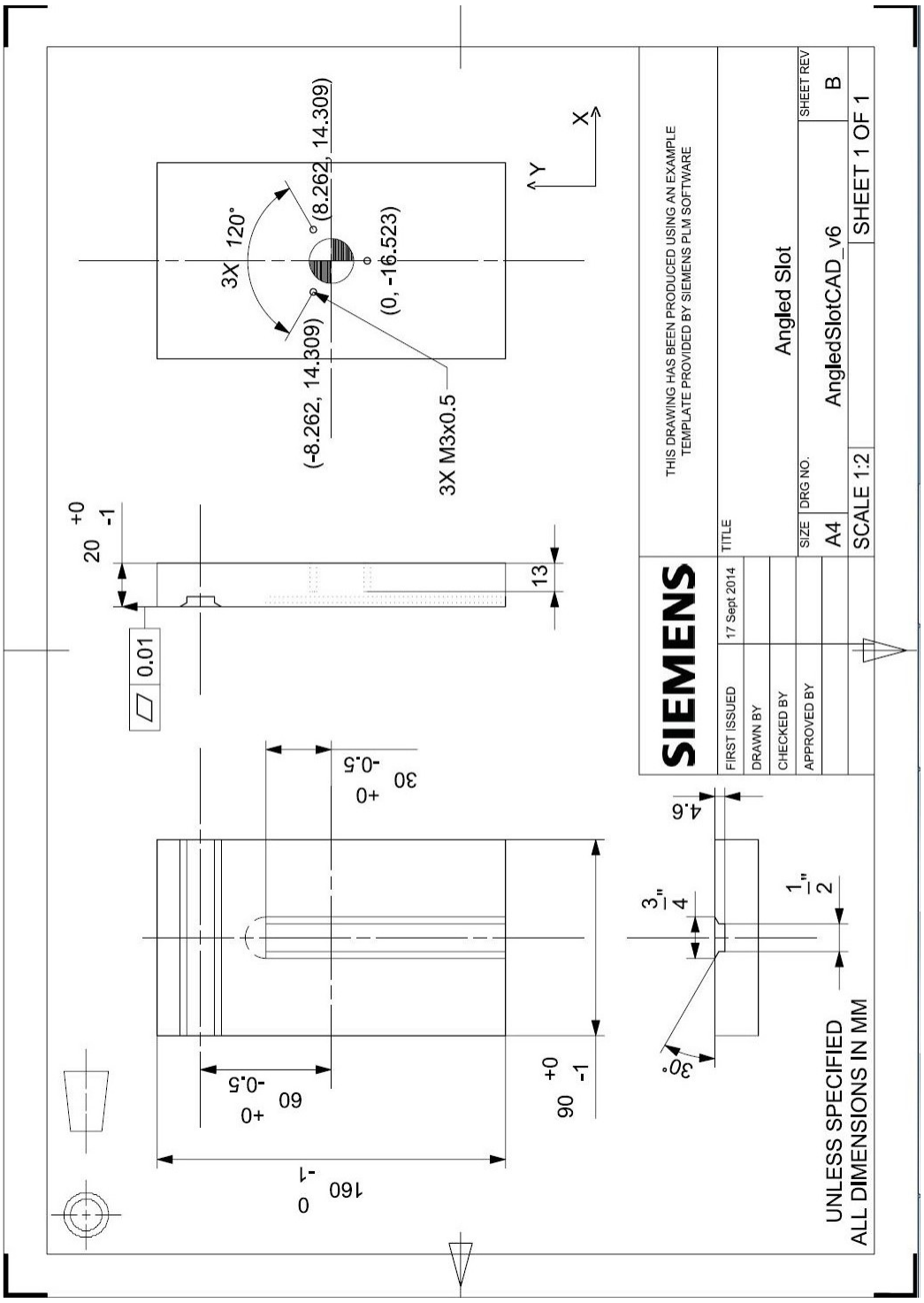


Figure B.1: Angled slot artefact CAD drawing

Appendix C

Weighted Total Least Squares

This appendix shows the setup and implementation of the weighted parallel plane fit method discussed in [58].

C.1 kernel.cu

```
#include <iostream>
#include <fstream>
#include <string>
#include "CudaHeader.h"
#include "definitions.h"

void cpu_impl(XYZW ***, XYZW *, long *);

void largest_eigen(double *, double *, double *, int *);
```

```
void GenPlaneAB(XYZW ***, long *);

__global__ void WarmUpKernel() {}
__global__ void wt_cen_sum_kernel(XYZW *Points, BUNDLE *gpu_data, long
    num_pts, int plane_num) {
   __shared__ doubleXYZW shared_thread_sum[THREADS_PER_BLOCK];//allocate
       as many cells as threads
   unsigned long i = blockIdx.x * blockDim.x + threadIdx.x;

   int PointsPerThread = num_pts / (THREADS_PER_BLOCK*NUM_OF_BLOCKS);
   int RemainingPoints = num_pts % (THREADS_PER_BLOCK*NUM_OF_BLOCKS);
   int n_start = 0, n_end = 0;
   if (i < RemainingPoints){
      n_start = PointsPerThread*i + i;
      n_end = n_start + PointsPerThread + 1;
   }
   else
   {
      n_start = PointsPerThread*i + RemainingPoints;
      n_end = n_start + PointsPerThread;
   }

   doubleXYZW local_sum = { 0, 0, 0, 0 };

#pragma unroll //loop unrolling
```

```
for (int j = n_start; j < n_end; j++)
{
   local_sum.x += Points[j].w * Points[j].x;
   local_sum.y += Points[j].w * Points[j].y;
   local_sum.z += Points[j].w * Points[j].z;
   local_sum.w += Points[j].w;

}

//copying thread local sums to shared memory
shared_thread_sum[threadIdx.x].x = local_sum.x;
shared_thread_sum[threadIdx.x].y = local_sum.y;
shared_thread_sum[threadIdx.x].z = local_sum.z;
shared_thread_sum[threadIdx.x].w = local_sum.w;

__syncthreads(); // syncronizing threads

//summing threads-wise local sums of a block in the first thread
if (threadIdx.x == 0) //in the first thread
{
   doubleXYZW block_sum = { 0, 0, 0, 0 }; //local block sum; will
       internally use registers
   for (int i = 0; i < THREADS_PER_BLOCK; i++)
   {
      block_sum.x = block_sum.x + shared_thread_sum[i].x;
      block_sum.y = block_sum.y + shared_thread_sum[i].y;
```

```
        block_sum.z = block_sum.z + shared_thread_sum[i].z;
        block_sum.w = block_sum.w + shared_thread_sum[i].w;
      }

      atomicAdd(&(gpu_data[plane_num].cen_sum.x), block_sum.x);
      atomicAdd(&(gpu_data[plane_num].cen_sum.y), block_sum.y);
      atomicAdd(&(gpu_data[plane_num].cen_sum.z), block_sum.z);
      atomicAdd(&(gpu_data[plane_num].cen_sum.w), block_sum.w);

   }
}

//only upper diagonal of matrix S'S will be computed since it is symmetric
    3X3 matrix
__global__ void STS_kernel(XYZW *Points, BUNDLE *gpu_data, long num_pts,
    int plane_num) {
   __shared__ double shared_STS[THREADS_PER_BLOCK][6];//allocate as many
       cells as threads
   unsigned long i = blockIdx.x * blockDim.x + threadIdx.x;

   int PointsPerThread = num_pts / (THREADS_PER_BLOCK*NUM_OF_BLOCKS);
   int RemainingPoints = num_pts % (THREADS_PER_BLOCK*NUM_OF_BLOCKS);
   int n_start = 0, n_end = 0;
   if (i < RemainingPoints){
      n_start = PointsPerThread*i + i;
      n_end = n_start + PointsPerThread + 1;
```

```
}
else
{
   n_start = PointsPerThread*i + RemainingPoints;
   n_end = n_start + PointsPerThread;
}

double local_STS[6] = { 0, 0, 0, 0, 0, 0 }; //local copy of matrix S'S

for (int j = n_start; j < n_end; j++)
{

   //centeroid
   //calculating centeroid here instead of copying data back and forth
      from the gpu to cpu to perform the simple divison
   float gpu_wt_cen[3] = { 0, 0, 0 };
   gpu_wt_cen[X] = gpu_data[plane_num].cen_sum.x /
      gpu_data[plane_num].cen_sum.w;
   gpu_wt_cen[Y] = gpu_data[plane_num].cen_sum.y /
      gpu_data[plane_num].cen_sum.w;
   gpu_wt_cen[Z] = gpu_data[plane_num].cen_sum.z /
      gpu_data[plane_num].cen_sum.w;

   XYZW offset_point = { 0, 0, 0, 0 }; //point offset by centeroid
   int pt_index = j + plane_num*num_pts;
```

```
    offset_point.x = Points[j].x - gpu_wt_cen[X];
    offset_point.y = Points[j].y - gpu_wt_cen[Y];
    offset_point.z = Points[j].z - gpu_wt_cen[Z];

    //estaimting local S'S
    local_STS[0] += (Points[j].w) * (offset_point.x) * (offset_point.x);
        //multiplying the two square rooted w removes the root
    local_STS[1] += (Points[j].w) * (offset_point.x) * (offset_point.y);
    local_STS[2] += (Points[j].w) * (offset_point.x) * (offset_point.z);
    local_STS[3] += (Points[j].w) * (offset_point.y) * (offset_point.y);
    local_STS[4] += (Points[j].w) * (offset_point.y) * (offset_point.z);
    local_STS[5] += (Points[j].w) * (offset_point.z) * (offset_point.z);
}

//copying thread local S'S elements to shared memory
shared_STS[threadIdx.x][0] = local_STS[0];
shared_STS[threadIdx.x][1] = local_STS[1];
shared_STS[threadIdx.x][2] = local_STS[2];
shared_STS[threadIdx.x][3] = local_STS[3];
shared_STS[threadIdx.x][4] = local_STS[4];
shared_STS[threadIdx.x][5] = local_STS[5];

__syncthreads(); // syncronizing threads

//summing threads-wise local sums of a block in the first thread
if (threadIdx.x == 0) //in the first thread
```

```
    {
        double block_STS[6] = { 0, 0, 0, 0, 0, 0 }; //local block sum; will
            internally use registers
        for (int i = 0; i < THREADS_PER_BLOCK; i++)
        {
            block_STS[0] += shared_STS[i][0];
            block_STS[1] += shared_STS[i][1];
            block_STS[2] += shared_STS[i][2];
            block_STS[3] += shared_STS[i][3];
            block_STS[4] += shared_STS[i][4];
            block_STS[5] += shared_STS[i][5];

        }

        atomicAdd(&(gpu_data[plane_num].STS[0]), block_STS[0]);
        atomicAdd(&(gpu_data[plane_num].STS[1]), block_STS[1]);
        atomicAdd(&(gpu_data[plane_num].STS[2]), block_STS[2]);
        atomicAdd(&(gpu_data[plane_num].STS[3]), block_STS[3]);
        atomicAdd(&(gpu_data[plane_num].STS[4]), block_STS[4]);
        atomicAdd(&(gpu_data[plane_num].STS[5]), block_STS[5]);

    }

}

int main()
```

```
{
   long num_pts[NUM_PLANES]; //number of points in the planes

   XYZW *Gen[NUM_PLANES];
   XYZW **Plane_Arr[NUM_PLANES];

   for (int j = PLANE_A; j < NUM_PLANES; j++)//looping over planes A and B
   {
      num_pts[j] = 0;
      Gen[j] = NULL;
      Plane_Arr[j] = &Gen[j];
   }

   GenPlaneAB(Plane_Arr, num_pts);//returns data and the number of points

   std::ofstream f_out;
   f_out.open("Output.txt", std::ios::app);
   f_out << "\n" << NUM_PLANE_SETS << " parallel plane sets used\n";
   f_out << "Compiled in " << COMPILATION_MODE << " mode at " << __DATE__
       << " @ " << __TIME__ << "\n"; //shows only last compile time
   f_out << "Computation for " << NUM_PLANES << " planes each with " <<
       NUM_POINTS << " points\n";
   //print results
   f_out << "Smallest eigen value of STS and required normal \n";

   long i = 0;
```

```
cudaFree(0);

cudaStream_t stream[NUM_PLANES];

XYZW *Points[NUM_PLANES], *gpu_Points[NUM_PLANES]; //points on planes

for (int j = PLANE_A; j < NUM_PLANES; j++)//looping over planes A and B
{
   Points[j] = NULL;
   gpu_Points[j] = NULL;

   cudaStreamCreate(&stream[j]);

   HANDLE_ERROR(cudaMallocHost((XYZW**)&Points[j], MAX_SIZE_A *
       sizeof(XYZW)));
   HANDLE_ERROR(cudaMalloc((XYZW**)&gpu_Points[j], MAX_SIZE_B *
       sizeof(XYZW)));

   //copy data into a CPU varaible
   //in reality a more efficient data reading/memory mapping will be used
   for (i = 0; i < num_pts[j]; ++i) {

      (Points[j])[i].x = (*Plane_Arr[j])[i].x;
      (Points[j])[i].y = (*Plane_Arr[j])[i].y;
      (Points[j])[i].z = (*Plane_Arr[j])[i].z;
```

```
      (Points[j])[i].w = (*Plane_Arr[j])[i].w;

   }
}

BUNDLE *cpu_data = NULL, *gpu_data = NULL;//centeroid sum and sts
    matrix data
HANDLE_ERROR(cudaMallocHost((BUNDLE**)&cpu_data,
    NUM_PLANES*sizeof(BUNDLE)));
HANDLE_ERROR(cudaMalloc((BUNDLE**)&gpu_data,
    NUM_PLANES*sizeof(BUNDLE)));

WarmUpKernel << <NUM_OF_BLOCKS, THREADS_PER_BLOCK >> >();
HANDLE_ERROR(cudaGetLastError());

// computataions on all plane sets
for (int plane_set_index = 0; plane_set_index < NUM_PLANE_SETS;
    plane_set_index++)
{

   //code organised so that points are copied are while everything else
       happens

   for (int j = PLANE_A; j < NUM_PLANES; j++)//looping over planes A and
       B
   {
```

```
//initialization
cpu_data[j].cen_sum.x = 0;
cpu_data[j].cen_sum.y = 0;
cpu_data[j].cen_sum.z = 0;
cpu_data[j].cen_sum.w = 0;
for (int i = 0; i < 6; i++) cpu_data[j].STS[i] = 0;

HANDLE_ERROR(cudaMemcpyAsync(&gpu_data[j], &cpu_data[j],
    sizeof(BUNDLE), H2D, stream[j]));

//using async here and copying each plane seperately to hide
    latency
HANDLE_ERROR(cudaMemcpyAsync(gpu_Points[j], Points[j],
    (num_pts[j]) * sizeof(XYZW), H2D, stream[j]));

//launch kernel
wt_cen_sum_kernel << <NUM_OF_BLOCKS, THREADS_PER_BLOCK, 0,
    stream[j] >> >(gpu_Points[j], gpu_data, num_pts[j], j);

HANDLE_ERROR(cudaGetLastError());

STS_kernel << <NUM_OF_BLOCKS, THREADS_PER_BLOCK, 0, stream[j] >>
    >(gpu_Points[j], gpu_data, num_pts[j], j);
HANDLE_ERROR(cudaGetLastError());

//finally copy the plane A and B summed STS from the gpu
```

```
    HANDLE_ERROR(cudaMemcpyAsync(cpu_data[j].STS, gpu_data[j].STS, 6 *
        sizeof(double), D2H, stream[j]));

} //end of iterations on both planes

//synchronize streams
for (int j = 0; j < NUM_PLANES; j++) {
    cudaStreamSynchronize(stream[j]);
}

double STS[6] = { 0, 0, 0, 0, 0, 0 };

//summing up STS results from both the planes
for (int i = 0; i < 6; i++)
    for (int j = 0; j < NUM_PLANES; j++)
        STS[i] += cpu_data[j].STS[i];

//finding largest eigen value / vector for the STS
double LEvec[3] = { 0, 0, 0 }, LEval = 0, temp = 0;
int iter_num = 0;
largest_eigen(STS, LEvec, &LEval, &iter_num);

//creating matrix A whose largest eigen value corresponds to the
    smallest eigen value of STS with the same eigen vector
//A = 2 * lambda*I - STS;
double A[6] = { 0, 0, 0, 0, 0, 0 };
```

```
    A[0] = 2 * LEval - STS[0];
    A[1] = -STS[1];
    A[2] = -STS[2];
    A[3] = 2 * LEval - STS[3];
    A[4] = -STS[4];
    A[5] = 2 * LEval - STS[5];

    temp = LEval;

    // the eigen vector result obtained is the required normal
    largest_eigen(A, LEvec, &LEval, &iter_num);

    //smallest eigen value of STS
    temp = 2 * temp - LEval;

    f_out << "GPU :\t" << temp << "\t" << LEvec[0] << "\t" << LEvec[1] <<
        "\t" << LEvec[2] << "\n";

} //end of plane sets

//free GPU memory
HANDLE_ERROR(cudaFree(gpu_data));
HANDLE_ERROR(cudaFreeHost(cpu_data));

//synchronize streams
for (int j = 0; j < NUM_PLANES; j++) {
```

```
    //free points memory
    HANDLE_ERROR(cudaFreeHost(Points[j]));
    HANDLE_ERROR(cudaFree(gpu_Points[j]));
    //destroy streams
    cudaStreamDestroy(stream[j]);
}
HANDLE_ERROR(cudaDeviceReset());

for (int plane_set_index = 0; plane_set_index < NUM_PLANE_SETS;
     plane_set_index++)
{
    //call the cpu implementation here
    XYZW cpu_eigen = { 0, 0, 0, 0 };

    cpu_impl(Plane_Arr, &cpu_eigen, num_pts);

    f_out << "CPU :\t" << cpu_eigen.w << "\t" << cpu_eigen.x << "\t" <<
        cpu_eigen.y << "\t" << cpu_eigen.z << "\n";

}//end of plane sets

f_out.close();

////free input data
for (int j = 0; j < NUM_PLANES; j++) {
    free(Gen[j]);
```

```
   }
   return 0;
}
```

C.2 Header (CudaHeader.h)

```
//C++ headers
#include <iostream>
#include <fstream>
#include <string>

// CUDA headers
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#define H2D cudaMemcpyHostToDevice
#define D2H cudaMemcpyDeviceToHost

#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))

// Handle error is copy of a commonly used error handle obtained from
    other sources
void HandleError(cudaError_t err, const char *file, int line)
{
   if (err != cudaSuccess) {
```

```
      printf("%s in %s at line %d\n", cudaGetErrorString(err), file, line);
      getchar(); // to prevent output window from closing immediately after
          error display
      exit(1);
   }
}

//double atomic add taken from CUDA documentation
__device__ double atomicAdd(double* address, double val) {
   unsigned long long int* address_as_ull = (unsigned long long
       int*)address;
   unsigned long long int old = *address_as_ull, assumed;
   do {
      assumed = old; old = atomicCAS(address_as_ull, assumed,
          __double_as_longlong(val + __longlong_as_double(assumed)));
      // Note: uses integer comparison to avoid hang in case of NaN (since
          NaN != NaN)
   } while (assumed != old);
   return __longlong_as_double(old);
}
```

C.3 Header (definitions.h)

```
typedef struct {
```

```
   float x;
   float y;
   float z;
   float w; //weight
}XYZW;

typedef struct {
   double x;
   double y;
   double z;
   double w; //weight
}doubleXYZW;

typedef struct {
   doubleXYZW cen_sum; //centeroid sum
   double STS[6];// STS (symmetric) matrix elements row by row
}BUNDLE; //structure to bundle other parameters to tranfer to GPU

//definitions to imrove readbility
#define X 0
#define Y 1
#define Z 2
#define W 3

#define PLANE_A 0
#define PLANE_B 1
```

```
#define NUM_POINTS 1000000 //should be greater than number of cores

#define THREADS_PER_BLOCK 192 //threads per block ; the number has to be
    32 for gtx 480
#define NUM_OF_BLOCKS 15 //number of blocks
#define EPSILON 1E-5 //ERROR EPSILON
#define TOL 1E-6

#define NUM_PLANE_SETS 1 // number of plane sets used

#define NUM_PLANES 10 // number of parallel planes per set

# define NUM_ITER 1
```

C.4 Header (largest eigen.cpp)

```
//C++ headers
#include <iostream>
#include <fstream>
#include <string>
#include "definitions.h"

//calculates the largest eigen value-vector pair using power method
void largest_eigen(double *A, double *LEvec, double *LEval, int *iter_num)
```

```
{
   double v1[3] = { 1, 0, 0 };
   double v0[3] = { 1, 0, 0 };
   float rel_err = 10; //initialization
   double lambda = 0;
   int iter = 0;
   while (rel_err > TOL)
   {
      //    v0 = v1
      v0[0] = v1[0];
      v0[1] = v1[1];
      v0[2] = v1[2];

      //    v1 = A * v0;
      v1[0] = A[0] * v0[0] + A[1] * v0[1] + A[2] * v0[2];
      v1[1] = A[1] * v0[0] + A[3] * v0[1] + A[4] * v0[2];
      v1[2] = A[2] * v0[0] + A[4] * v0[1] + A[5] * v0[2];

      //    lambda = norm(v1); // norm(v0) is already 1 so no need to divide
      lambda = sqrt(v1[0] * v1[0] + v1[1] * v1[1] + v1[2] * v1[2]);

      // normalizing     v1 = v1 / norm(v1)
      v1[0] = v1[0] / lambda;
      v1[1] = v1[1] / lambda;
      v1[2] = v1[2] / lambda;
```

```
    //   rel_err = norm(v1 - v0); % norm(v0) is already 1 so no need to
        divide

    rel_err = sqrt((v1[0] - v0[0]) * (v1[0] - v0[0]) + (v1[1] - v0[1]) *
        (v1[1] - v0[1]) + (v1[2] - v0[2]) * (v1[2] - v0[2]));
    iter++;
  }

  //Copy results
  LEvec[0] = v1[0];
  LEvec[1] = v1[1];
  LEvec[2] = v1[2];

  *LEval = lambda;
  *iter_num = iter;
}
```

Bibliography

[1] Aguirre-Cruz, J. A. and Raman, S. (2005). Torus form inspection using coordinate sampling. *Journal of manufacturing science and engineering*, **127**(1), 84–95.

[2] Ahn, S. J. (2004). *Least squares orthogonal distance fitting of curves and surfaces in space*, volume 3151. Springer Science & Business Media.

[3] Ahn, S. J., Rauh, W., Cho, H. S., and Warnecke, H.-J. (2002). Orthogonal distance fitting of implicit curves and surfaces. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **24**(5), 620–638.

[4] American Society of Mechanical Engineers (2009). Dimensioning and Tolerancing. ASME Y14.5-2009, ASME.

[5] Andrea, D. (2005). Punching and stamping: How to compare production cost, technical report, dallan rollformers and systems. Technical memo, Dallan S.p.A., Treviso, Italy.

[6] Barbero, B. R. and Ureta, E. S. (2011). Comparative study of different digitization techniques and their accuracy. *Comput. Aided Des.*, **43**(2), 188–206.

[7] Bernal, C., de Agustina, B., Marín, M. M., and Camacho, A. M. (2014). Accuracy analysis of fridge projection systems based on blue light technology. *Key Engineering Materials*, **615**.

[8] Besl, P. J. and McKay, N. D. (1992). Method for registration of 3-d shapes. In *Robotics-DL tentative*, pages 586–606. International Society for Optics and Photonics.

[9] Bosché, F. (2010). Automated recognition of 3D CAD model objects in laser scans and calculation of as-built dimensions for dimensional compliance control in construction. *Advanced engineering informatics*, **24**(1), 107–118.

[10] Bräuer-Burchardt, C., Heist, S., Kühmstedt, P., and Notni, G. (2014). High-speed 3d surface measurements with a fringe projection based optical sensor. In *Proc. of SPIE Vol*, volume 9110, pages 91100E–1.

[11] Carter, J. A., Tucker, T. M., and Kurfess, T. R. (2008). 3-Axis CNC path planning using depth buffer and fragment shader. *Computer-Aided Design and Applications*, **5**, 612–621.

[12] Che, C. and Ni, J. (2000). A ball-target-based extrinsic calibration technique for high-accuracy 3-d metrology using off-the-shelf laser-stripe sensors. *Precision Engineering*, **24**(3), 210–219.

[13] Checkmate (2015). *version 12.1*. Origin International Inc., Richmond Hill, ON, Canada.

[14] Dolcemascolo, D. (2006). *Improving the extended value stream: lean for the entire supply chain*. Productivity Press.

[15] Double edged cutter (2013). KBC Tools, 6200 Kennedy Rd, Mississauga, ON Canada L5T 2Z1.

[16] Erdõs, G., Nakano, T., and Váncza, J. (2014). Adapting CAD models of complex engineering objects to measured point cloud data. *CIRP Annals - Manufacturing Technology*, **63**, 157–160.

[17] FaroArm (2013). FARO, 250 Technology Park Lake Mary, FL 32746, USA.

[18] Feng, H., Liu, Y., and Xi, F. (2001). Analysis of digitizing errors of a laser scanning system. *Journal of the International Societies for Precision Engineering and Nanotechnology*, pages 185–191.

[19] Forbes, A. (1991). Least-squares best-fit geometric elements. *NPL report DITC*.

[20] Geomagic Qualify (2010). 3D Systems Inc., Cary, NC, USA.

[21] Gocator (2013). *Snapshot sensor*. LMI Technologies, Delta, BC, Canada.

[22] Guennebaud, G., Jacob, B., *et al.* (2010). Eigen v3. http://eigen.tuxfamily.org.

[23] Günther, C., Kanzok, T., Linsen, L., and Rosenthal, P. (2013). A GPGPU-based pipeline for accelerated rendering of point clouds. *Journal of WSCG*, **21**, 153–161.

[24] He, W., Li, Z., Zhong, K., Shi, Y., Zhao, C., and Cheng, X. (2014). Accurate and automatic extrinsic calibration method for blade measurement system integrated by different optical sensors. In *SPIE/COS Photonics Asia*, pages 92761D–92761D. International Society for Optics and Photonics.

[25] Hocken, R. J. and Pereira, P. H. (2012). *Coordinate measuring machines and systems*. CRC Press.

[26] Hu, X., Li, X., and Zhang, Y. (2013). Fast filtering of LiDAR point cloud in urban areas based on scan line segmentation and GPU acceleration. *IEEE Geoscience and Remote Sensing Letters*, **10**(2), 308–312.

[27] InnovMetric Software Inc (2013). 3D metrology hardware review. Technical memo, InnovMetric Software Inc, Qubec, Canada.

[28] International Organization for Standardization (2010a). Geometrical Product Specifications (GPS)-Dimensional Tolerancing-Part 1: Linear Sizes. ISO 14405-1:2010, ISO.

[29] International Organization for Standardization (2010b). Geometrical Product Specifications (GPS)-Dimensional Tolerancing-Part 3: Angular Sizes. ISO 14405-3:(emerging), ISO.

[30] Inui, M., Umezu, N., and Shimane, R. (2015). Shrinking sphere: A parallel algorithm for computing the thickness of 3D objects. *Computer-Aided Design and Applications*, pages 1–9.

[31] Kamil, M., Kamely, A., Saifudin, H., Puvanasvaran, A., and Dan, M. (2010). The source of uncertainty in 3D laser scanner. *Journal of Advanced Manufacturing Technology*.

[32] Kinsner, M. (2011). *Close-range machine vision for gridded surface measurement*. Phd thesis, McMaster University.

[33] Kinsner, M., Spence, A., and Capson, D. (2010). Gpu accelerated sheet forming grid measurement. *Computer-Aided Design and Applications*, **7**(5), 675–684.

[34] Kovacs, L., Zimmermann, A., Brockmann, G., Baurecht, H., Schwenzer-Zimmerer, K., Papadopulos, N. A., Papadopoulos, M. A., Sader, R., Biemer, E., and Zeilhofer, H. F. (2006). Accuracy and precision of the three-dimensional assessment of the facial surface using a 3-D laser scanner. *IEEE Transactions on Medical Imaging*, **25**(6), 742–754.

[35] Krishnamurthy, A., McMains, S., and Haller, K. (2011). GPU-accelerated minimum distance and clearance queries. *IEEE Transactions on Visualization and Computer Graphics*, **17**(6), 729–742.

[36] Krystek, M. and Anton, M. (2007). A weighted total least-squares algorithm for fitting a straight line. *Measurement Science and Technology*, **18**(11), 3438.

[37] Kurella, V., Stone, B., and Spence, A. (2017). GPU accelerated CAD to inspection data deviation colormap generation. *Computer-Aided Design and Applications*, **14**(2), 234–241.

[38] Kurfess, T. R., Tucker, T. M., Aravalli, K., and Meghashyam, P. (2007). GPU for CAD. *Computer-Aided Design and Applications*, **4**(6), 853–862.

[39] Lee, R. S. and Ren, M. K. (2011). Development of virtual machine tool for simulation and evaluation. *Computer-Aided Design and Applications*, **8**(6), 849–858.

[40] Lei, X., Song, H., Xue, Y., Li, J., Zhou, J., and Duan, M. (2011). Method for cylindricity error evaluation using geometry optimization searching algorithm. *Measurement*, **44**(9), 1556–1563.

[41] Martínez, S., Cuesta, E., Barreiro, J., and Álvarez, B. (2009). Analysis of laser scanning and strategies for dimensional and geometrical control. *The International Journal of Advanced Manufacturing Technology*, **46**(5-8), 621–629.

[42] Martínez, S., Cuesta, E., Barreiro, J., and Álvarez, B. (2010). Methodology for comparison of laser digitizing versus contact systems in dimensional control. *Optics and Lasers in Engineering*, **48**(12), 1238–1246.

[43] MATLAB (2014). *version R2014a*. The MathWorks Inc., Natick, Massachusetts.

[44] Mekid, S. and Luna, H. (2007). Error propagation in laser scanning for dimensional inspection. *International Journal of Metrology*.

[45] Mohan, P., Shah, J., and Davidson, J. K. (2013). A library of feature fitting algorithms for GD&T verification of planar and cylindrical features. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages V02AT02A005–V02AT02A005. ASME.

[46] Mohan, P., Haghighi, P., Shah, J. J., and Davidson, J. K. (2015). Development of a library of feature fitting algorithms for CMMs. *International Journal of Precision Engineering and Manufacturing*, **16**(10), 2101–2113.

[47] Morey, B. (2013). New metrology culture improving Chrysler quality. `http://www.sme.org/MEMagazine/Article.aspx?id=77027`.

[48] Ozan, Ş. and Gümüştekin, Ş. (2013). Calibration of double stripe 3D laser scanner systems using planarity and orthogonality constraints. *Digital Signal Processing*.

[49] Park, S.-Y., Choi, S.-I., Kim, J., and Chae, J. S. (2011). Real-time 3D registration using gpu. *Machine Vision and Applications*, **22**(5), 837–850.

[50] Peiravi, A. and Taabbodi, B. (2010). A reliable 3D laser triangulation-based scanner with a new simple but accurate procedure for finding scanner parameters. *Journal of American Science*, **6**(5).

[51] Ram, M. P. M., Kurfess, T. R., and Tucker, T. M. (2008). Least-squares fitting of analytic primitives on a GPU. *Journal of Manufacturing Systems*, **27**(3), 130–135.

[52] Rashidizad, H. and Rahimi, A. (2014). Building three-dimensional scanner based on structured light technique using fringe projection pattern. *Journal of Computing and Information Science in Engineering*, **14**(3), 035001.

[53] Renishaw AM1 adjustment module (2014). Renishaw plc, Wotton-under-Edge, Gloucestershire, UK.

[54] Renishaw touch probe (2013). Renishaw plc, Wotton-under-Edge, Gloucestershire, UK.

[55] Sanders, J. and Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional.

[56] Shakarji, C. M. (1998). Least-squares fitting algorithms of the NIST algorithm testing system. *Journal of Research-National Institute of Standards and Technology*, **103**, 633–641.

[57] Shakarji, C. M. (2011). Coordinate measuring system algorithms and filters. In

J. Fagerberg, D. Mowery, and R. Nelson, editors, *Coordinate Measuring Machines and Systems, Second Edition*, page 153182. CRC Press.

[58] Shakarji, C. M. and Srinivasan, V. (2013). Theory and algorithms for weighted total least-squares fitting of lines, planes, and parallel planes to support tolerancing standards. *Journal of Computing and Information Science in Engineering*, **13**(3), 031008.

[59] Spence, A., Capson, D., Sklad, M., Chan, H.-L., and Mitchell, J. (2008). Simultaneous large scale sheet metal geometry and strain measurement. *Journal of manufacturing science and engineering*, **130**(5).

[60] Srinivasan, V., Shakarji, C. M., and Morse, E. P. (2012). On the enduring appeal of least-squares fitting in computational coordinate metrology. *Journal of Computing and Information Science in Engineering*, **12**(1), 011008.

[61] Wang, Y., Li, L., Ni, J., and Huang, S. (2009). Form tolerance evaluation of toroidal surfaces using particle swarm optimization. *Journal of Manufacturing Science and Engineering*, **131**(5), 051015.

[62] Watkins, D. S. (2004). *Fundamentals of matrix computations*, volume 64. John Wiley & Sons.

[63] Xi, F., Liu, Y., and Feng, H.-Y. (2001). Error compensation for three-dimensional line laser scanning data. *The International Journal of Advanced Manufacturing Technology*, **18**(3).

[64] Xue, K., Kurella, V., and Spence, A. (2016). Multi-sensor blue LED and touch

probe inspection system. *Computer-Aided Design and Applications*, **13**(6), 827–834.

[65] Zexiao, X., Jianguo, W., and Qiumei, Z. (2005). Complete 3d measurement in reverse engineering using a multi-probe system. *International Journal of Machine Tools and Manufacture*, **45**(12), 1474–1486.