12-2007

# Least Squares Fitting of Analytic Primitives on a GPU

Meghashyam Panyam mohan ram
*Clemson University*, mpanyam@clemson.edu

# LEAST SQUARES FITTING OF ANALYTIC PRIMITIVES ON A GPU

---

A Thesis
Presented to
the Graduate School of
Clemson University

---

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Mechanical Engineering

---

by
Meghashyam Panyam
December 2007

---

Accepted by:
Dr. Thomas Kurfess, Committee Chair
Dr. Thomas Tucker
Dr. Joshua Summers

# ABSTRACT

Metrology systems take coordinate information directly from the surface of a manufactured part and generate millions of (X, Y, Z) data points. The inspection process often involves fitting analytic primitives such as sphere, cone, torus, cylinder and plane to these points which represent an object with the corresponding shape. Typically, a least squares fit of the parameters of the shape to the point set is performed. The least squares fit attempts to minimize the sum of the squares of the distances between the points and the primitive. The objective function however, cannot be solved in the closed form and numerical minimization techniques are required to obtain the solution. These techniques as applied to primitive fitting entail iteratively solving large systems of linear equations generally involving large floating point numbers until the solution has converged. The current problem in-process metrology faces is the large computational times for the analysis of these millions of streaming data points. This research addresses the bottleneck using the Graphical Processing Unit (GPU), primarily developed by the computer gaming industry, to optimize operations.

The explosive growth in the programming capabilities and raw processing power of Graphical Processing Units has opened up new avenues for their use in non-graphic applications. The combination of large stream of data and the need for 3D vector operations make the primitive shape fit algorithms excellent candidates for processing via a GPU. The work presented in this research investigates the use of the parallel processing capabilities of the GPU in expediting specific computations involved in the fitting procedure. The least squares fit algorithms for the circle, sphere, cylinder, plane, cone and torus have been implemented on the GPU using NVIDIA’s Compute Unified Device Architecture (CUDA). The implementations are benchmarked against those on a CPU which are carried out using C++. The Gauss Newton minimization algorithm is used to obtain the best fit parameters for each of the aforementioned primitives. The computation times for the two implementations are compared. It is demonstrated that the GPU is about 3-4 times faster than the CPU for a relatively simple geometry such as the circle while the factor scales to about 14 for a torus which is more complex.

# DEDICATION

This thesis is dedicated to God, my father, who now lives with Him, my mother and my sister, the two people without whom this work would not be possible.

# ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Thomas R. Kurfess, for his priceless guidance, undying faith and support throughout the course of this project. Dr. Kurfess' constant feedback and high expectations have driven me to keep making progress and complete this work. I would like to thank my other advisor, Dr. Thomas M. Tucker, for all the valuable inputs and discussions which have played a significant role in the completion of my work. I extend my sincere thanks to Dr. Joshua Summers for being a part of my committee and the helpful discussions. A special thanks to my fellow graduate students Koushik Aravalli and Santosh Tiwari for their selfless help during the course of this work. I would also like to thank my roommates Nitendra Nath, Sourabh Patki and Parth Bhavsar and all my colleagues at office for their constant encouragement and help. Finally, I would like to thank my family and friends for all the love and care without which this work would be incomplete.

This work was partially funded by the National Science Foundation under Grant Number DMII-0600902. The government has certain rights in this material. Any opinions, findings and conclusions or recommendations are those of the author and do not necessarily reflect the views of the National Science Foundation.

# TABLE OF CONTENTS

Table of Contents (Continued)

# LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

List of Figures (Continued)

# CHAPTER I

# INTRODUCTION

## Traditional Metrology

Metrology is the science of measurement. It is a very important aspect of the design and manufacturing process. Any part is manufactured based on a set of desired characteristics it should possess which are specified by the designer It is required to have a quantification of how well the part or artifact meets the design requirements. This is generally done by the inspection process or measurement which involves procuring the dimensional characteristics of the part and comparing it with the design.

Traditionally, manufactured parts are measured based on specific features such as diameter, length and flatness using instruments such as calipers, micrometers and other gauges. This measurement however is one or two-dimensional and neglects the three dimensional characteristics of the part. Further, this kind of measurement is susceptible to human error as it is not automated. The continuous quality improvement has given rise to the need for a new generation of metrology tools. To this end, a variety of metrology systems have been developed to take 3D coordinate information directly from the surface of the part and the process is termed as Coordinate Metrology.

## Coordinate Metrology

Coordinate metrology or computational metrology provides more complete information of the manufactured parts. This process involves the application of mathematical models or tools to solve problems encountered in metrology. Generally these problems involve the analysis of coordinate measurements made from a Coordinate Measuring Machine (CMM) or any other measuring system such as theodolites, laser tracking systems and the like. There are a wide variety of approaches to analyze the data and determine whether the points in the data set match the intended geometry described in the CAD file. Currently, state-of-the-art measuring devices generate millions of data points and their analysis requires complicated procedures and algorithms. This analysis often requires comparison of these large number of data points to thousands of surfaces represented by the CAD model to register or localize the points to it by doing a least squares fit. The other problem that arises is the need to fit a primitive shape such as a plane, sphere, cone, cylinder or torus to a collection of large 3D data measured from an object of the corresponding shape. The solution is the least squares fit of the parameters of the shape to the point set. In both the problems mentioned above, the major concern is the speed with which the required analysis or computations can be carried out. This problem has been addressed previously by the development of novel analytical routines which have provided significant speed increases. However, the development of new higher speed hardware, has further pushed the limits of analytical tools and hence there is a need for software to analyze the data accurately and efficiently.

## Research Objective

The work presented in this research addresses the bottleneck in the least squares fitting of geometric primitives to coordinate data by using currently available hardware, a Graphical Processing Unit (GPU). Recently, GPUs have undergone an evolution to powerful and flexible processing units. They have been shown to provide substantial gain in processing time in areas other than graphics such as general purpose as well as scientific computation including fluid flow simulation, finite element analysis and many other applications.

The least squares fit for analytic primitives to 3D data involve construction of a function by determining the distance from each point in the data set to the surface in contention. Generally, this function cannot be solved in the closed form and requires numerical techniques such as the Newton method. This method requires computation of the first derivative of the distance function with respect to the minimization parameters along with other arithmetic intensive 3D vector operations. These qualities of the problem make it an excellent candidate for processing via a GPU.

The goal of this research is to identify and implement specific computations involved in the least squares minimization algorithms of analytic primitives viz. circle, sphere, cylinder, plane, cone and torus on the GPU. Furthermore, these implementations are benchmarked against CPU implementations of the same to investigate the potential gain in processing time.

## Thesis Outline

The thesis is organized as explained in this section. Chapter 2 presents prior work in the field of least squares minimization techniques, coordinate metrology and the use of GPUs in various engineering and general purpose applications. Chapter 3 provides an insight on the hardware and software architecture used in this work. Chapter 4 describes the mathematical formulation of the algorithm, the C++ and GPU implementations for each primitive mentioned. The next chapter includes the comparison of results and discussions. Chapter 7 includes conclusions of the work.

# CHAPTER II

# LITERATURE REVIEW

## Coordinate Metrology and Fitting

Hopp was one of the first to coin the term ‘Computational Metrology’. In his article, he addresses aspects of metrology such as fitting objectives and their implementation in data analysis software and a procedure to test coordinate measuring system (CMS) software. He applied least squares fit and extremal fit objectives to a circle fitting problem in the presence of perturbations (measurement errors) and compared the fits. He confirmed that extremal fits propagate more of the point measurement error than least squares fit. The article mainly identifies problems with fitting software and proposes a ‘Black Box’ approach to test the analysis software. The testing method is limited to supplying the procedure with the fitting problems and analyzing the fit results[1].

Lin *et al.* compare fitting algorithms for the Coordinate Measuring Machine (CMM). They benchmarked four algorithms namely least squares methods, minimax max-deviation method, minimum average deviation method and the convex hull method. The algorithms are evaluated with respect to tolerance zone size, solution uniqueness and computational efficiency. They concluded that the least squares has numerical qualities that make it robust and useful in computational metrology even though they do not explicitly state that any particular method is best for all CMM data analysis [2].

Choi in his dissertation addresses computational analysis of 3D measurement data which involves evaluating geometric dimensions from the data and verification of conformance to tolerances. He formulated problems for both least squares fit and extreme fits. He states that extreme fits are useful because they conform to tolerance theory. Choi also investigated the uncertainty associated with dimensional evaluation. He concluded that evaluation uncertainty is mainly due to the stochastic noise that dominates in least squares fits and the sampling uncertainty in the case of extreme fits[3]. He applied statistical theories in terms of confidence regions for estimated parameters to formalize the uncertainty.

Gass *et al.* investigated the problem of analyzing CMM data taken against circular (spherical) features of manufactured parts. In this paper, they describe a linear programming approach for the algebraic Chebychev formula to determine reference circles and spheres and related tolerance annuluses. They compare the solutions obtained with the algebraic least squares solutions and conclude that this method yields concentric circles whose separation is less than that of the corresponding least squares solution [4].

Shakarji and Clement describe reference algorithms developed at the National Institute of Standards and Technology to fit geometric shapes to data using Chebychev, maximum-inscribed and minimum-circumscribed criteria. Using an improved, approach they develop more reliable reference algorithms for Chebychev fitting of lines, planes, circles, spheres, cylinders and cones. For each of these, they obtain the fit through an iteration that begins using a least squares fit and then refine it to the desired Chebychev fit. They outline the steps taken for each geometric shape to reduce the number of fit parameters thus improving the performance of their algorithms. They document their test results and demonstrate that their algorithms perform better than the algorithms found in industrial use [5].

Hopp and Levenson discuss the importance of the performance of fitting software used in a Coordinate Measuring Systems to evaluate the geometric characteristics of manufactured parts. They lay out a set of criteria in developing performance measures for testing the software developed by the National Institute of Standards and Technology (NIST). Seven geometry types are considered, namely, line, circle, plane, sphere, cylinder, cone and torus. The procedure involves collection of data sets for each geometry type, generating a fit for each data set, called the "Reference Fit" using NIST-developed fitting algorithms, and comparing this to a "Test Fit" generated by the software. The differences between each pair of fits are represented by a set of "Difference Parameters". A statistical approach is used to interpret the difference parameters as a performance measure. They also perform an uncertainty analysis of the software to provide quantitative measures of performance[6].

In 1997 Zwick outlined the applications of Orthogonal Distance Regression (ODR) which connotes a form of non-linear least squares regression in coordinate metrology [7]. He presents a few problems such as fitting geometric elements such as lines planes, cones, cylinders and parametric surfaces such as B-splines and NURBS surfaces and discusses the procedures involved in utilizing the variants of ODR explicit, implicit and parametric in solving these problems.

## Least Squares Fitting Techniques

Various techniques have been developed for least squares fitting of curves, surfaces and geometric primitives (Plane, Cube, Sphere and Torus) to a set of 3D data points. Pratt developed direct least squares methods namely exact fit, simple fit and spherical fits which require roughly the work of matrix inversion or extraction of Eigen values. All these methods consider the algebraic distance from a point in the data set to the surface. He showed that the primitives when considered as algebraic surfaces rather than conventional parametric lend themselves directly to least squares techniques as naturally as parametric surfaces. The exact fit method gives a good solution of approximately fitting a shape to n-1 points in a set of n points. But this method is computationally expensive for large number of points as the matrix set up is first triangularized then the co-factors are computed to obtain the polynomial approximating the surface to be fit. Also the method does not treat the case where the rank of the matrix is less than n-1 whence the points underdetermine the shape. The simple fit method uses the Cholesky decomposition to obtain an upper triangular matrix which is then treated in the same method as in the exact fit method. The computational cost using this increases drastically when the number of points increases. Furthermore, the quality of fit obtained by applying this technique has not been analyzed. The spherical fit method overcomes the problem of obtaining a bad fit as the curve of best fit approaches a line. But, for scattered data, the algorithm is not very efficient in that the invariant is larger for outlying data which causes the fit to be more responsive and hence decreases the curvature of fit. Another drawback of this method is that it involves the extraction of Eigen vectors [8].

Taubin (1991) develops least square fitting techniques by representing the approximate distance (Minimum Mean Squared Approximate Distance) from the points to the surface by implicit equations. He replaces the original implicit function with a new one whose value is a better approximate of the distance. The problem of fitting curves and surfaces is the minimization of the approximate mean square distance which can be reduced to the generalized Eigen vector computation [9].

Keren and Gotsman provide a novel approach in the use of a family of parameterized implicit polynomials for fitting star-shaped curves and surfaces (2D and 3D). In their paper, they discuss the advantages and disadvantages of using implicit polynomials as modeling tool in fitting of surfaces approximated by them. They develop two methods ‘Line Convexity’ and ‘Focus of Expansion’ to force the polynomial to be convex (function of one variable) to overcome the pathologies in fit such as loops and holes [10].

Blane, Lei, Civi and Cooper show that the conventional Eigen value/ Eigen vector and the Minimum Mean Squared Euclidean Distance methods provide unfaithful fits when the data are not accurately represent able by a polynomial. They develop a new algorithm called the “3L Algorithm”. They present a method which involves setting up a surface whose value is the Euclidean distance from a point on it to the closest point in the data set. They then implement the least squares fit of this surface with an explicit polynomial. They show that their algorithm can handle 2D curves and 3D surfaces represented by 16th and 10th degree polynomial respectively and claim that it yields physically meaningful representations even when the object is too complex to be fit exactly by the degree polynomial used [11].

Michael Plass and Maureen Stone present a method for fitting shapes defined by a discrete set of data points with a parametric piecewise cubic polynomial curve. Their goal was to give an efficient representation for graphic arts. They developed two techniques, one being dynamic programming for determining the knot positions and the other an iterative method for fitting a parametric cubic with optional end point and tangent vector constraints to a set of data points [12].

Sourlier and Bucher developed an algorithm to standardize the procedure of data fitting. Their paper illustrates the unification of the process of fitting standard analytic primitives and parametric sculpted surfaces. They developed a single minimization routine based on the $L_2$ – Norm which operates independently of the surface function and arrives at the theoretical best fit. Thus they provide a modular program with subroutines for different geometries (i.e., plane, spheres, torus, splines) [13].

Alistair B. Forbes (1990) describe algorithms designed for the use of coordinate measuring systems (CMS), for finding the best fit geometric elements (lines, planes, circles, spheres, cylinders and cones) to metrological data. The first model he discusses is the isotropic model, where the residual errors are computed normal to the surface and the measurement errors in all directions are assumed to be equal and uncorrelated. He developed robust and efficient parameterizations and optimization algorithms for this model. The best fit line and plane are obtained by performing the Singular Value Decomposition (SVD) on the matrix whose columns are obtained by subtracting the centroid of the $x, y, z$ data points. Circle and sphere fitting procedures are analogous due to the fact that the residual error is formulated by computing the distance from the measured points and their centers and the resulting objective function is solved iteratively by using the Gauss-Newton or Newton minimization algorithm. The algebraic formulation of the residual error is solved to obtain the initial guesses for the two problems. He suggests a novel and efficient method to fit cylinders and cones, these geometries are translated and rotated such that they are in the standard position (aligned with the $z$-axis) before implementing the Gauss-Newton algorithm. This avoids the cumbersome calculations involved in formulating the Jacobian matrices [14]. Forbes then compared the results of the cylinder fit from this model with the second model namely the anisotropic model wherein the measurement error is unequal in the $x$, $y$ and $z$ directions. He summarizes that results from both models are similar but the anisotropic model has advantages in that it can be used in different CMS and also a wide variety surfaces and data fitting problems can be tackled using the routines based on this model.

Ames addresses the problem of fitting of sphere, right-circular cone and right-circular cylinders. He states that the traditional approach of taking partial derivatives of the objective function, equating them to zero and solving the equations is unstable due to bad numerical behavior. He employed the Singular Value Decomposition least squares to construct solid models from 3D data. The problem is formulated by setting up fixed functions called the Basis functions. These functions are solved by iterating through the terms, removing the zero term to construct reduced basis functions and eventually solve the system of equations [15].

Ahn *et al.* (2001) proposed simple non-parametric algorithms for the geometric fitting of circle/sphere and ellipse/hyperbola/parabola. Their algorithms are based on the coordinate description of the corresponding point on the geometric feature for a given point, where the connecting line of the two points is the shortest path from the given point to the geometric feature. They use the Gauss-Newton method to minimize the objective function. The initial parameter vector for the circle and sphere are given by the center of gravitation and the Root Mean Square (RMS) central distances [16].

Gander, Golub and Strebel compare the accuracy and computational cost of three types of fitting viz. algebraic distance minimization, geometric distance minimization and geometric distance fitting in parametric form as applied to the least squares fit of circles and ellipses. They conclude that though the algebraic solution is computationally cheaper by a factor of 10-100 compared to the other algorithms, it is quite unstable and inaccurate. Furthermore, they implemented non-linear least squares algorithms including Gauss-Newton, Newton, Gauss-Newton with Marquardt modification, variable projection (varpro) and orthogonal distance regression (odr) algorithms to compute the geometric fit and compare them with respect to stability and efficiency. They observed that while the Newton method is most efficient, when applied to the parameterized algorithm if the problem is well posed, 'odr' is competitive with algorithms specifically written for fitting ellipses and 'varpro' is computationally expensive as well as inefficient [17] .

Lukacs, Marshall and Bajcsky (1997) show that the straightforward algebraic methods developed by [8] work well when applied to fitting of spheres, for which, under suitable normalization, the minimized algebraic distance reflects the geometric distance, but these methods approximate the true geometric distance in an unfaithful way for other geometries. They show that the partial derivatives of their modified distance function with respect to any of the surface parameters and spatial parameters are the same as the original distance function. They provide methods for parameterization of analytic primitives in which the exact expression for the distance is replaced by a simplified function which is much easier to compute.  The technique of parameterization given by them is robust in the sense that as the principal curvature of the surfaces being fitted decreases, the results converge to surfaces which best describe the data. They propose a method to obtain initial estimates of the parameters by computing the rotational axis for each of the primitives (except sphere) based on the estimate of a surface normal vector. They also outlined the implementation of these techniques in the segmentation of data based on a recover and select paradigm [18].

Lukacs *et al.* validate their least squares fitting routines for geometric primitives and the segmentation strategy developed in [18] by implementation on simulated data as well as real data obtained from a laser scanner in the presence of noise. They provide results of these implementations and conclude that their methods work successfully in practical environments and are accurate for their intended tasks. Their paper demonstrates that their methods are robust and handle degeneracy in both estimating and fitting surfaces with very low curvature [19].

Shakarji (1998) developed and implemented least squares fitting algorithms for linear as well as non-linear geometries as reference software for the NIST's Algorithm Testing System. He presents the defining parameters, distance equations, the objective functions, their derivatives and normalizations required to fit planes, lines, spheres, cones, cylinders and tori. Langrange multipliers are used in solving the constrained minimization problems which are developed into the standard Eigen vector problems to fit linear geometries (planes).  The Levenberg-Marquardt algorithm is used to minimize the objective function for the other geometries [20].

Atieg and Watson address the problem of developing good numerical methods to fit a curve or surface to data in metrology and pattern recognition applications. The paper is concerned with a modification of the basic non-linear least squares problem in which the orthogonal distances from the data points to the curve or the surface may not always be defined. Analogues of the Gauss-Newton method are developed, analyzed and illustrated by 2D and 3D examples such as line, circular disc and non solid cylinder. The method adopted involves rotating the data set so that the object to be fit is aligned along a coordinate axes. They show that this simplifies the problem by effectively removing variables that can be defined at each iteration. This enables the direct calculation of the gradient of the objective function thus avoiding the need for second derivatives[21].

Atieg and Watson further survey and compare a class of particular methods to solve non-linear least squares problems and place the members of this class into a unified framework. They show that, of the two variants of the Gauss-Newton method developed, the one involving the computation of the second derivative is preferred over the method described earlier for fitting 3D curves as this avoids ill-conditioning of the problem. However, the first variant is preferred while fitting geometric elements because of the simplicity in computation and lesser number of iterations required for the solution to converge. They conclude that this method is strong when good initial guesses of the parameters are available [22].

Claudet presents the defining parameters and formulates the computation of deviation from the data points to the primitives viz. plane, sphere, cylinder, cone and torus. He discusses the differences between the three major minimization procedures viz. the Newton, Gauss-Newton and the Levenberg-Marquardt methods. He adopts the Levenberg-Marquardt method for minimization of the sum of squares of deviations. He verified and presented the results of parameter fitting of the geometries by selecting a reasonable initial value as input to the fitting routines [23].

Most least squares techniques mentioned require a good starting point or initial approximation for the minimization method to converge to a good fit. Kurfess and Chen developed Bounding Box strategies namely Axis Aligned and Minimum Bounding Box techniques to arrive at good initial guesses for Sphere, Plane, Torus, Line and Plane. The algorithms involved in these techniques are simple to implement and very fast [24].

## Graphical Processing Units (GPUs)

GPUs have evolved from being fixed function pipelines to fully programmable, floating point pipelines. They are highly optimized for fast rendering of geometries and image generation required in the gaming industry. They have a high memory bandwidth and support floating point arithmetic. The ability to reconfigure the graphics pipeline through shader programming coupled with parallel processing capabilities makes the GPU versatile. Recently, GPUs have been deployed in numerous non-graphic applications such as general purpose computing.

Olano and Lastra created a parallel graphics multi computer, Pixel Flow, using a shading language. This interactive graphics platform not only achieved very high frame rates but also could perform mathematical operations to calculate pixel values [25]. Thompson *et al.* indicated that GPUs can be used for purposes other than graphics rendering by developing a framework for solving general purpose problems on them. The framework is capable of accelerating regular operations on large vectors. They investigated this by applying the framework to matrix multiplication operations [26].

Kruger and Westermann described a general framework for the implementation of numerical simulation techniques on GPUs. They provided a novel approach by considering matrices as diagonal or column vectors and by representing vectors as 2D texture maps which considerably accelerated matrix vector and vector-vector operations. They also provided a GPU implementation of the conjugate gradient method to numerically solve 2D wave equation [27].

Buck *et al.* present Brook, a system of general-purpose computation on GPUs. They provide a compiler and a runtime system that maps the Brook language onto the existing programmable GPU Application Programming Interfaces (APIs). They implemented three different implementations on Brook to evaluate the performance. Their initial implementation included a vector scale and sum operations, dense matrix vector product followed by a scaled vector add. The other two implementations include a segmentation algorithm followed by a Fast Fourier Transform (FFT) application. They compare the results with corresponding CPU implementations. They achieve considerably greater performance with their implementation being 7 and 4.7 times faster than the CPU for the matrix operations and segmentation algorithms respectively. However the FFT was found to be only marginally quicker than its CPU counterpart. The major drawback of the system developed by them is that it does not allow the render-to-texture operation [28].

Galoppo *et al.* developed a novel algorithm to solve a dense linear system of equations on the graphics card to exploit its inherent parallelism. They implemented the LU decomposition algorithm with varying complexities and the results were benchmarked against the highly optimized ATLAS implementation on the CPU. Their implementation outperformed the CPU [29].

Jung implemented the Cholesky decomposition on the GPU to solve symmetric and positive definite matrices. This has been done based on a Primal-Dual Interior-Point Method with Brooks GPU (a stream programming model). He performs the Cholesky decomposition in two forms, the inner product and outer product form and compares their performance in two environments. He demonstrates the interior point method is best suited for implementation on the computational resources of the GPU [30].

Hoff *et al.* present a method for rapid computation of discrete Voronoi diagrams in two and three dimensions using graphics hardware. Their paper describes techniques developed for creating mesh of the distance function for each site with bounded error and how the mesh computes the Voronoi diagram rapidly. Their application is aimed at effectively solving the problem of finding collision free path for a moving robot [31].

Rumpf and Strzodka demonstrated a GPU finite element implementation to solve the linear heat equation and the anisotropic diffusion problem in image processing. Their basic idea of computing used the blending capacities and texture environment functions and its extensions to perform algebraic operations on images and also to optimize operations to reduce the number of rendering passes. They utilize the texture memory to store the initial, intermediate and final data of calculations. The paper illustrates the representation of the 2D grid of the finite element discretizations in the form of vectors on the graphics hardware on which numerical schemes operate. The Jacobi and conjugate gradient iterations are used to solve the system of linear equations.  Their implementation of the Jacobi solver achieved 300 MOP/s which outperforms any software implementation on the CPU [32].

Wu *et al.* solve a 2D fluid flow problem completely on the GPU. They adopted Semi-Lagrangian method to solve the Navier Stokes Equations to obtain real-time fluid effects. They adopted a method in which the scalar and vector variables are packed into four channels of texels. Taking into account the arbitrary boundary conditions, they group the grid nodes into different types according to their positions relative to obstacles and search the node that determines the value of the current node. The texture coordinate offsets are then computed according to the type of boundary condition of each node to determine the corresponding variables. They demonstrate that the GPU performs considerably faster than the CPU as the grid size is increased [33].

Hillesland *et al.* cast non-linear optimization as a data streaming process that is well matched to modern GPUs. They develop a framework capable of solving large non-linear optimizations concurrently and use it to solve image-based modeling problems, the light field mapping approximation of surface light fields and fitting the Lafortune model to spatial bidirectional reflectance distribution functions. The article describes the implementation of two optimization algorithms viz. Conjugate Gradient and the Steepest Descent algorithms and their implementation in solving the problems addressed. They observed that the Conjugate Gradient achieves much better convergence in a few iterations when compared to the Steepest Descent method. Furthermore, comparison between the CPU and GPU implementations indicated that the latter is about 5 times faster [34].

Computer-Aided Design applications stand to gain substantial benefits from the raw computational power of the GPU. McMains, Khardekar and Burton were the first ones to use GPUs in manufacturing analysis and design feedback. They present hardware accelerated algorithms to test 2D moldability of geometric parts and assist part design. These algorithms efficiently identify and graphically display undercuts as well as minimum and insufficient draft angles. They make use of the depth buffer to store the distance to the visible facet for each pixel. The efficiency of their algorithms lies in the fact that they identify groups of directions to determine whether they are undercut-free or not. They compared the GPU implementation with the commercial Solid Works face based undercut highlighting and indicate that the commercial results are not only far less informative about the exact location of undercuts  but also take 600 times longer to calculate and display [35].

Gray, Ismail and Bedi present a graphics hardware assisted approach to 5-Axis surface machining that builds upon a tool positioning strategy named the Rolling Ball Method. The depth buffer of the GPU is used to compute the data needed for this method which generates a gouge-free 5-axis curvature matched tool positions. In their implementation, some aspects of computation for tool positioning utilize the GPU; namely computation of Shadow Checking Grid Points' spatial displacements which are used to compute the pseudo-radius for each pixel from which the final radius of the rolling ball is selected. They outline that the graphics assisted approach eliminates the need for parameterization of the surface to be machined, thus allowing the machining of multiple patch triangulated surfaces [36].

A mechanistic model based on an adaptive and local depth buffer to calculate milling forces when machining a part on a multi-axis milling machine has been developed by Roth *et al.* Their paper presents a novel method of calculating chip geometry and volume of material removed during machining in order to determine the cutting forces. The terms “adaptive” and “local” depth buffers are used as the depth buffer is changed to be constantly aligned with the tool axis and sized to the tool instead of the work piece respectively. Previous tool positions are rendered to the scene and the depth buffer is saved. The current tool position is then rendered and the depth buffer is saved again to obtain the in-process chip geometry, which is the difference between the two states of the depth buffer. However, this model is limited to only flat end mills and inefficient as most of the depth buffer holds previous tool positions as it is sized to cover the tool [37]. The implementation has been modified to overcome the limitations of the model. The algorithm is updated to handle more complex tool shapes. The depth buffer is sized to the cutter teeth, thereby improving the memory requirements resulting in efficient usage [38].

Pabst *et al.* presented the concept of extended graphics pipeline that allows the rendering of complex primitive such as parametric and implicit surfaces. Their pipeline adds an intersection stage between the rasterization stage and the fragment program. The intersection stage reconstructs corresponding ray from the viewpoint for each fragment generated from the rasterization unit and computes the intersection with the surface contained in the bounding volume of the object to be displayed. The intersection point and the normal are passed into the fragment program. This integration into the graphics pipeline combines its high efficiency with the advantages of ray casting. They address the direct real-time rendering of trimmed NURBS surfaces. They use the Newton iteration for the intersection test and Iterative Bezier Clipping for exact trimming of the NURBS surfaces [39].

# CHAPTER III

# BACKGROUND ON GPU'S

## Overview

Graphical processing units (GPU's) have undergone an evolution from fixed-function processors to powerful, fully programmable, floating point pipelines. They are highly optimized for fast rendering of geometric primitives and image generation in computer gaming. Recently, they have been used for applications beyond graphics such as general purpose computation as well as scientific computation as discussed in the previous chapter. One of the main reasons for this explosive growth in the processing power of the GPU has been the increased interest in computer gaming which has pushed technology to the limit. The GPU is specialized for arithmetic intensive, parallel computations required in graphics rendering and is therefore designed such that more transistors are devoted to data processing rather than memory management or flow control. It is well suited for problems that require instructions be executed on large data sets at the same time.

Until now, the major issues that prevented access to all of the GPU's computational capabilities were [40]:

- A graphics API was required to program the hardware resulting in an overhead of learning the API as well as a high learning curve to the novice.
- GPU programs could read (*gather*) data from the hardware memory also called DRAM, but could not write (*scatter*) data to any part of the memory which resulted in a loss of programming flexibility.

- The DRAM memory bandwidth was low resulting in bottlenecks and under utilization of the available computation power.

This chapter describes a novel hardware architecture and programming model developed by NVIDIA® which provides a solution to the aforementioned problems and exposes the GPU as a parallel computing device.

## CUDA Architecture

CUDA stands for Compute Unified Device Architecture. It is a new hardware and software architecture that avoids the need for a graphics API to program the GPU to perform as a parallel computing device. The schematic below illustrates the software stack of this model.

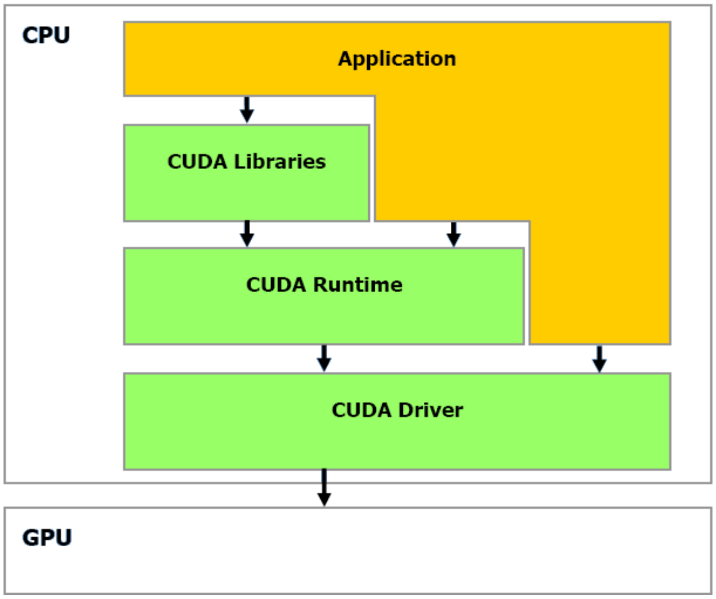

Figure 1: CUDA Software Stack [40]

The driver, API and its runtime and two high-level mathematical libraries form the software stack. The two highly optimized and efficient mathematical libraries supported by CUDA are the CUFFT-Fast Fourier Transform and the CUBLAS- Basic

Linear Algebra Subprograms. The CUDA API is an extension to the C programming language which allows for high programming flexibility as well as ease of learning. The read and write access operations to the DRAM of the card are extremely fast as CUDA features a memory space called the shared memory which brings data closer to the Arithmetic Logic Units (ALU) making them independent of memory bandwidth.

The GPU (device) can be programmed to operate as a multi-threaded co-processor to the CPU (host) to execute compute intensive portions of an application via CUDA. The portion of an application that needs to be executed many times on different data elements can be off-loaded to the GPU as an instruction set written in the form of a program called the *kernel* which is executed on the card as different threads. These threads are organized as a grid of thread blocks. A thread block consists of a batch of threads that access data from the shared memory and execute instructions in parallel. The maximum number of threads that can be specified per block is 512 for a G80. Synchronization points can be specified within a kernel program to coordinate memory accesses of these threads in runtime. Every thread within a block is addressed by a thread ID. A block can be one, two or three dimensional. A grid of thread blocks consists of a number of blocks that execute the same kernel and these are arranged in two dimensions. The main advantage is that the number of threads that can be launched in a single call to the kernel increase significantly by specifying a greater number of blocks. However, threads within one block cannot communicate with those in another resulting in reduced thread cooperation. Blocks are also addressed by their IDs within a grid. This model allows kernels to be efficiently executed on various devices with different parallel capabilities.

## **Memory Model**

GPU memory is primarily divided into six memory spaces namely:

- Registers
- Local memory
- Shared memory
- Global memory
- Constant memory
- Texture memory

The registers and local memories are per thread and the global, constant and texture memory spaces are per-grid while the shared memory space is per-block. The host or the CPU can read from or write to the constant, global and texture memory spaces while thee registers, local and shared memory spaces can be accessed only from the device. The global, constant and texture memory spaces are optimized for different memory usages. Figure 2 illustrates the memory model based on which a thread has access to the device memory through the set of memory spaces described in this section.

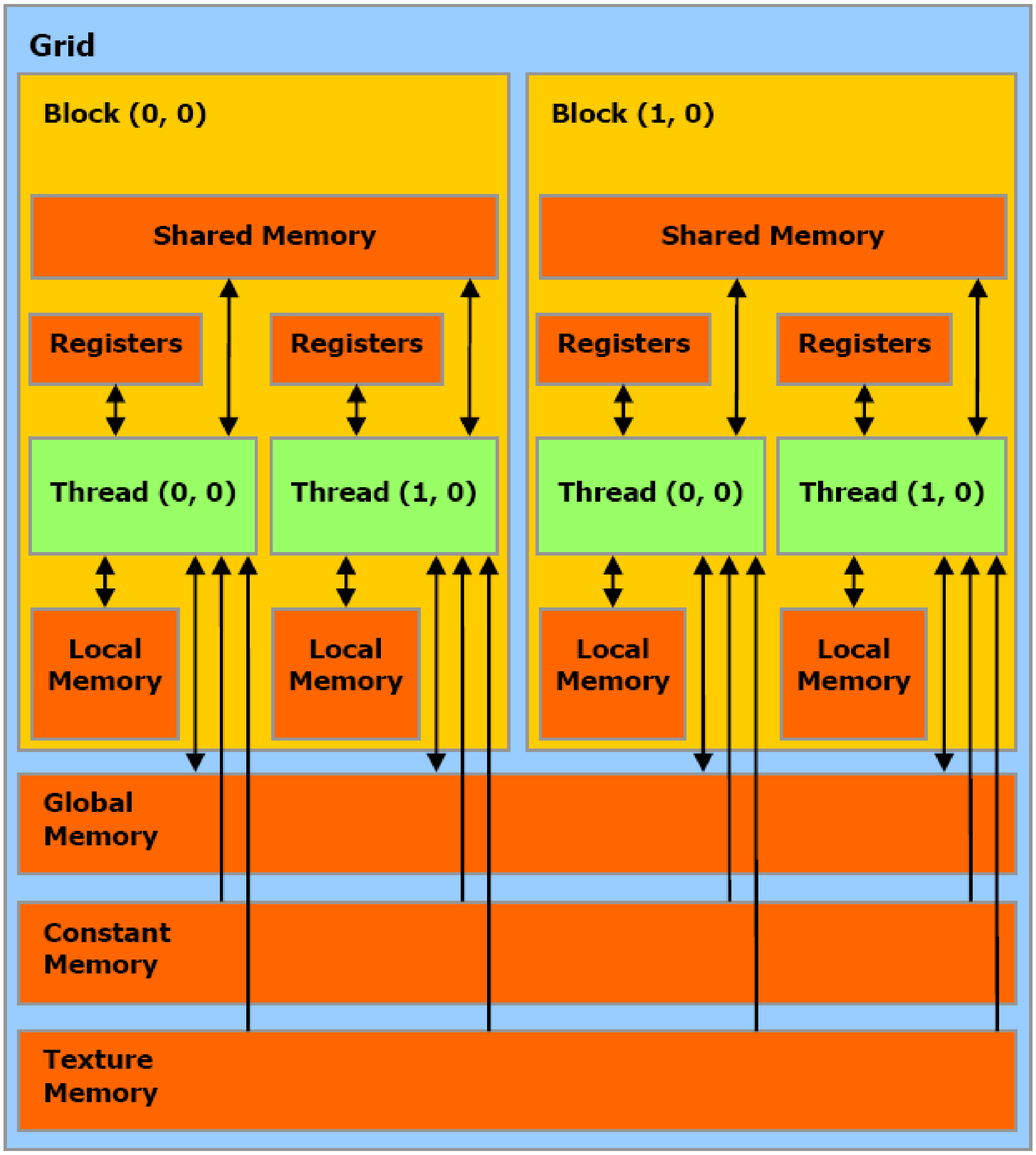

Figure 2: CUDA Memory Model

## Hardware Implementation

For all practical purposes, the GPU is implemented as set of multiprocessors each of which executes the same instruction on different data elements. In other words, each multiprocessor has Single Instruction Multiple Data (SIMD) architecture. The multiprocessor has four types of on-chip memory as shown in Figure 3 [40]; a set of 32-

bit local registers per processor, a shared memory that is shared by all processors, a read-only constant cache that speeds up reads from the constant memory space and a read-only texture cache that speeds up reads from the texture memory space . The constant and texture caches are also shared by all the processors.

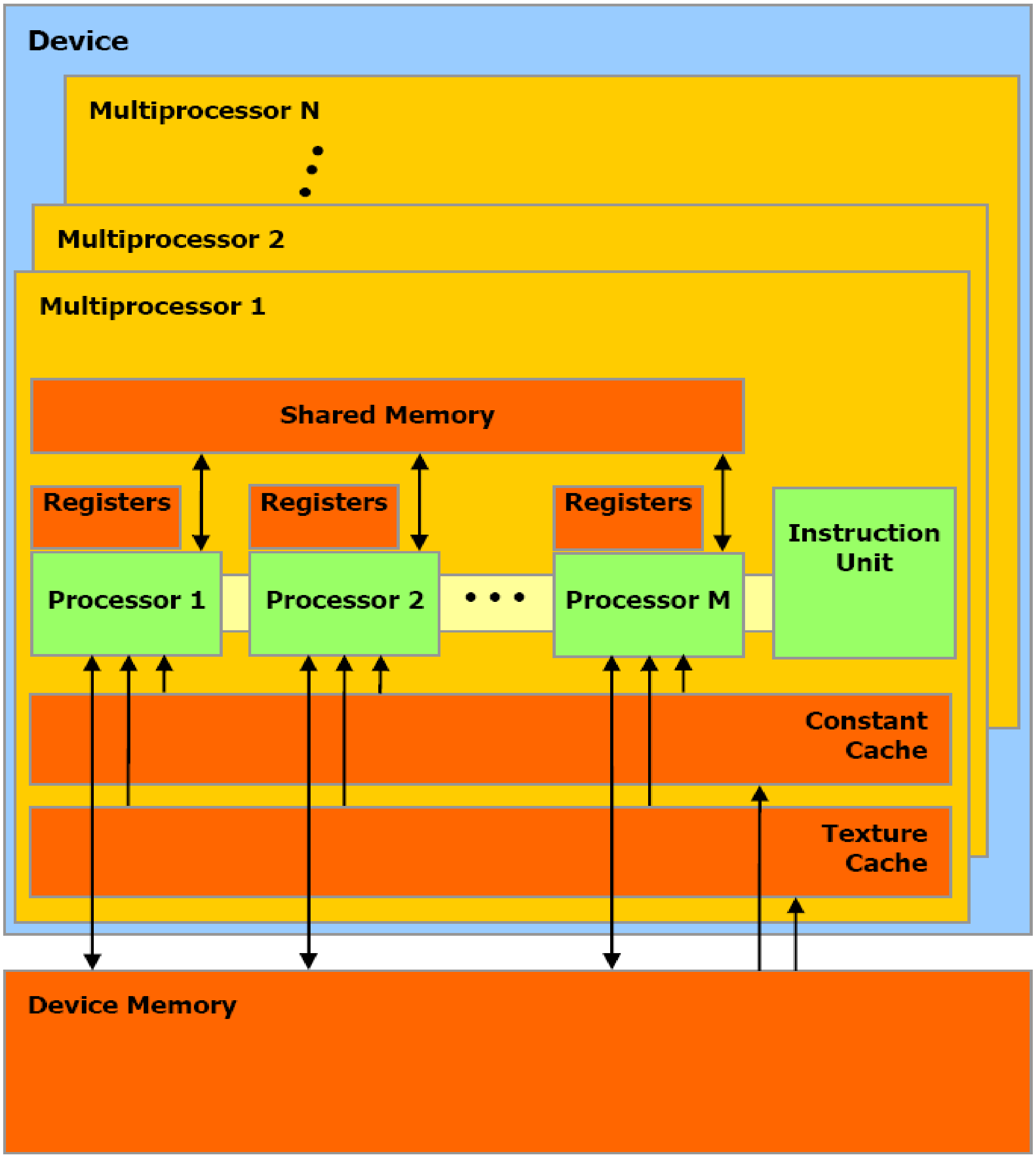

Figure 3: Hardware Model

The execution – A kernel is executed as grid of blocks in such a manner that one or more blocks are executed by each multiprocessor. The threads within a block are split into SIMD groups of equal thread size called warps. These are executed by a multiprocessor in SIMD fashion and hence it is necessary to ensure that all threads within a warp execute the same arithmetic instruction. A multiprocessor can execute several blocks in parallel by distributing the sets of registers in an efficient manner among them.

This research employs the mentioned novel architecture and programming model to save significant computation time in the least squares fits of analytic primitives by identifying and implementing specific computations involved in the non-linear minimization algorithms.

# CHAPTER IV

# FORMULATION AND IMPLEMENTATION

## Introduction

This chapter describes the mathematical formulation and the implementation of algorithms used for the least squares fit of six analytic geometry types namely circle, sphere, cylinder, plane, cone and torus on the CPU as well as the Graphical Processing Unit (GPU). The CPU implementation is done in C++ while the implementation on the graphics hardware is done using NVIDIA Corporation's novel software architecture CUDA. The first element required for the analysis is coordinate data which represents the geometries mentioned. The procedure for the generation of test data for all primitives is presented.

The least squares fit of all geometries except the plane is a non-linear minimization problem. The field of numerical optimization is rich in minimization techniques that can be made use of for finding the least squares solution or the "Best Fit". Techniques based on the Newton's method are considered to be the efficient in terms of speed and accuracy. The Gauss-Newton method is used to minimize the objective functions for all geometries except the plane. Computations from this algorithm that can be solved in parallel are identified and their implementation on the graphics hardware is explained. The graphics hardware implementation for the circle and sphere are similar while there are modifications for the cylinder, cone and torus. Further, the computation implemented on the hardware for a plane is different from all of the above and this is explained in this chapter.

## Point cloud generation

This section describes the method used to generate test data for all six of the above mentioned primitives. Coordinate data representing the geometry of all primitives is generated using MATLAB. All the codes written provide the flexibility of translating or rotating primitive data to any position in coordinate space. Suitable transformation matrices are included. Further more, the density of data generated can be specified by the user. The data generated for all primitives is written into suitable files for ease of analysis.

### Circle

For our implementation, the $x, y$ coordinates for a circle in a plane are generated by defining a vector θ in the interval $0, 2\pi$ and specifying

$$x = r * \cos\theta \tag{4.1}$$

$$y = r * \sin\theta \tag{4.2}$$

where $r$ is the radius of the circle. This procedure generates the coordinates for a circle which has its center at the origin.

### Sphere

The $x, y, z$ coordinates are generated by defining two vectors $\theta$ and $\phi$ in the interval [0,2π] specifying

$$x = r * \cos\theta * \sin\phi \tag{4.3}$$

$$y = r * \sin\theta * \sin\phi \tag{4.4}$$

$$z = r * \cos\theta \tag{4.5}$$

where $r$ is the radius of the sphere. The coordinates for the surface of the sphere obtained with this procedure are such that the center is at the origin. In other words, the sphere is centered about the origin. Figure 1 below shows a sample data set generated as compared to an ideal model of the sphere.

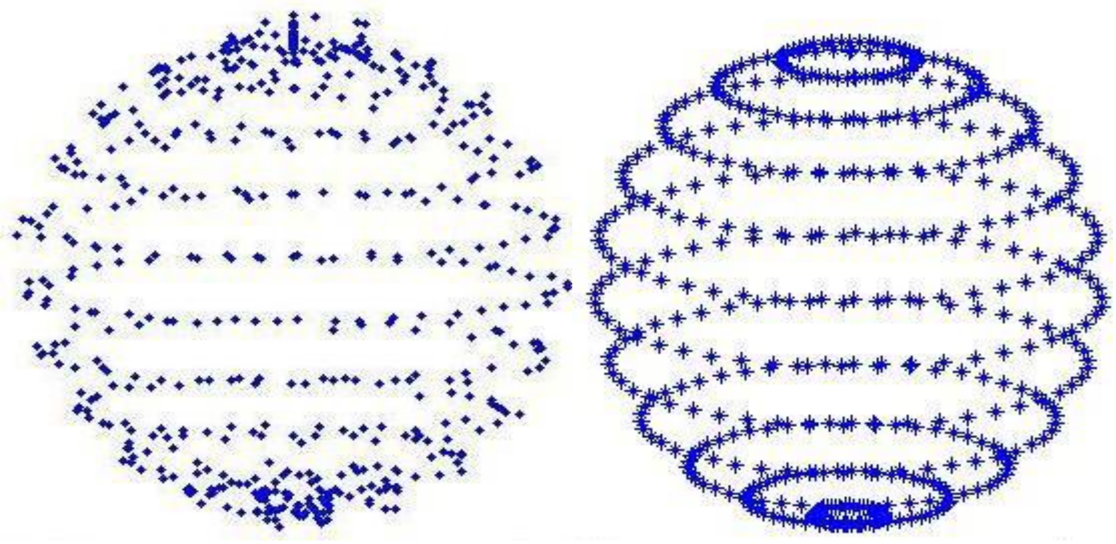

Figure 4: Test data compared to an ideal model (Sphere)

**Plane**

The parametric equation for a plane in 3D space is given by

$$Ax + By + Cz + D = 0 \tag{4.6}$$

Where the vector $A, B, C$ forms the normal to the plane. Figure 5 shows a typical X-Y plane. The normal to this plane is given along the $z$-axis as shown in the figure. Similarly, if the plane is in any arbitrary

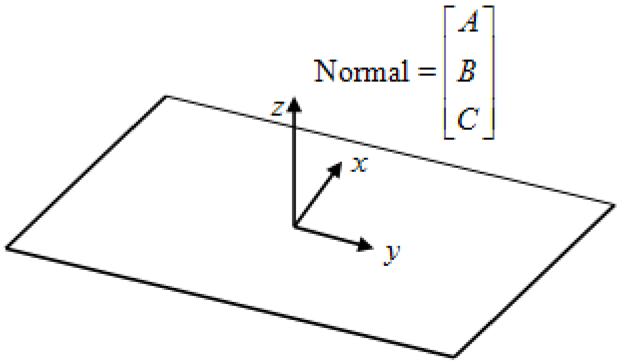

Figure 5: X-Y Plane

Three dimensional coordinate data for a plane for our implementation are generated by specifying the length and width of the plane in $x$ and $y$ directions respectively. The desired number of points along each axis is specified and the length and width are divided into vectors of corresponding sizes. The $z$-coordinate is constant for all $x$ and $y$ coordinates. The data can be translated and rotated to any position using suitable transformation matrices.

**Cylinder**

The parametric equations for the $x$ and $y$ coordinates of a cylinder are the same as those specified for the circle. In general, a cylinder can be of infinite length and hence a length needs to be specified for completeness. A typical right circular cylinder with length $l$ and radius $r$ aligned with the $z$ - axis is as shown in Figure 6.

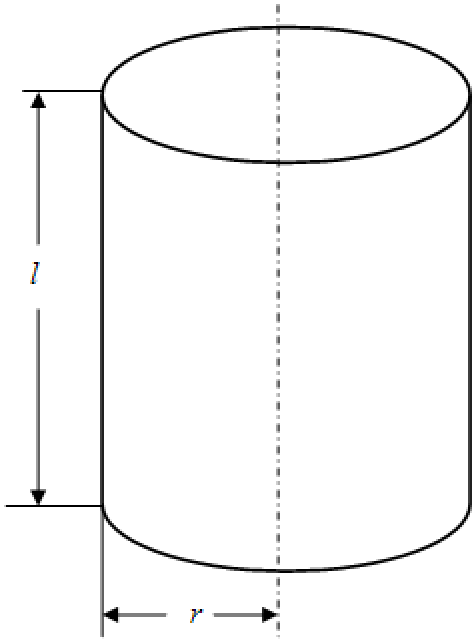

Figure 6: Typical right-circular cylinder

3D coordinate data for the cylinder are generated by specifying $x$ and $y$ coordinates as given by equations (4.1) and(4.2) respectively. The $z$-coordinates are specified such that they are constant for one set of $x$ and $y$-coordinates. This is done by dividing the length into a vector of size same as that of the vector $\theta$. In other words, the cylinder data are a stack of circles (of desired radius) aligned along the vertical axis with spacing between them depending on the length required.

**Cone**

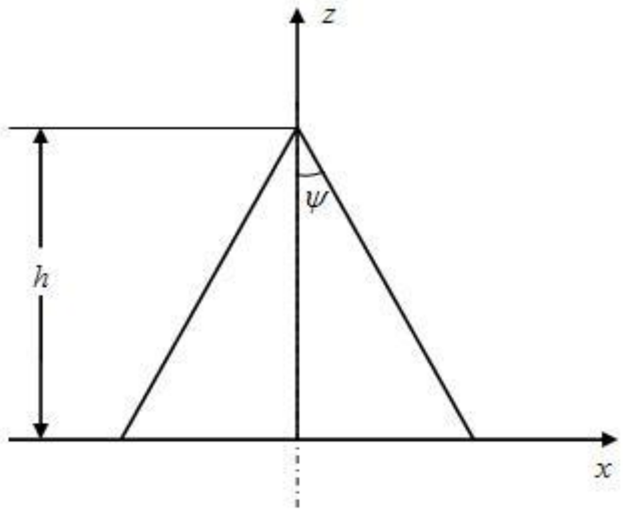

Figure 7: Cross-section of a cone aligned with the $z$ axis

The parametric equations for a cone having axis of symmetry on the $z$ - axis and apex at any height $h$ above the origin with the apex is given by the following

$$x = \frac{h-z}{h} * r * \cos\theta \tag{4.7}$$

$$y = \frac{h-z}{h} * r * \sin\theta \tag{4.8}$$

Where, $r$ is the radius of the base of the cone, $h$ is the height at which the apex is positioned, $z$ is the desired step from the vector in the interval $0, h$ and $\theta$ is a vector in the interval $0, 2\pi$ . In the more general case, a cone can be defined only by its apex-semi angle ($\psi$) by positioning the apex at the origin. The height or the base radius are not necessary parameters. However, either of these parameters is specified for completeness of the geometry as shown in Figure 7.

In this implementation, the $x$, $y$ and $z$-coordinates for a cone are specified in the similar manner as that for a cylinder. The $z$-coordinates are specified by dividing the height of the cone into a vector of size equal to the number of rings desired in the cone. Another vector $\theta$ of size equal to the number of points desired in each ring is defined from $0, 2\pi$ .The radius is computed at each $z$-coordinate and the $x$ and $y$ coordinates are specified in the same way as given by equations (4.1) and (4.2). Two different types of data sets of varying sizes are generated for the cone. In the first type the, points generated include the apex of the cone while the latter does not consider the presence of the apex. This is done to address two different scenarios faced in the least squares fit of a cone which are discussed in the following section.

**Torus**

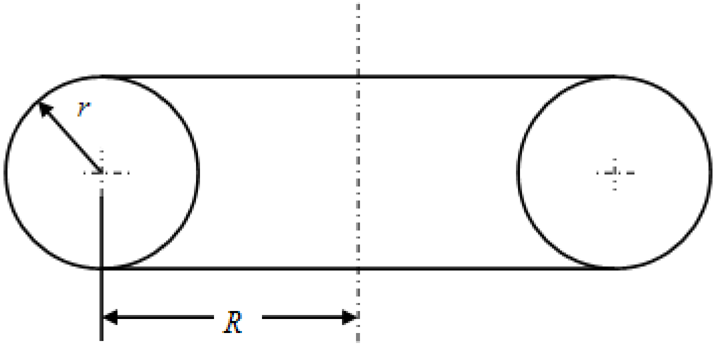

Figure 8: Cross-section of a standard torus

Torus is a geometry which is more commonly known as the “donut”. The cross section of a standard torus is as shown in Figure 8. The parametric equations for the $x$, $y$ and $z$-coordinates of this torus are given by

$$x = R*\cos\theta + r*\cos\phi*\cos\theta \tag{4.9}$$

$$y = R * \sin\theta + r * \cos\phi * \sin\theta \qquad (4.10)$$

$$z = r * \sin\phi \qquad (4.11)$$

Where, $R$ is the major radius, $r$ is the minor radius. If the major radius is less than or equal to the minor radius ($R \leq r$), the torus degenerates into a spindle torus or a sphere respectively. The data in our case are generated based on these parametric equations. Two vectors $\theta$ and $\phi$ are defined from $0, 2\pi$ .

**Noise**

Noise is induced in all coordinate data that are generated in the form of a random number that is added to the $x$, $y$ and $z$ coordinates. However, the random numbers generated are normally distributed with a mean of zero and a standard deviation which is a fraction of one. In the case of the circle which is 2D, noise is added while computing the $x$ and $y$ coordinates as explained earlier. For the 3D case, it is added to all the three coordinate data. Figure 9 shows the effect of adding noise to a data point in 3D space.

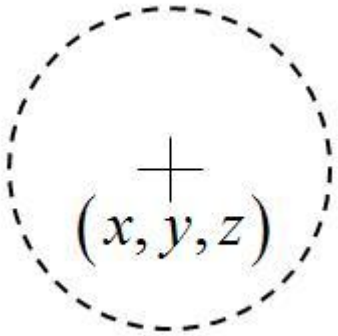

Figure 9: Noise added to a 3D point

The normally distributed noise added to any point in 3D results in the point being displaced to any position within the circle shown in the figure.

**Transformations**

Homogenous transformations are an essential part of point cloud generation. In general data sampled off a manufactured part are in an arbitrary position in 3D space. To simulate a practical situation, a provision is made to transform the data generated to any desired position. A $4 \times 4$ homogenous transformation matrix consisting of three rotations $R_x$, $R_y$, $R_z$ and three translations $T_x$, $T_y$ and $T_z$ for the three coordinate axes is set up. The array consisting of the coordinate data for a particular primitive is multiplied by this to obtain the transformed data. However, in the case of the circle (2D), the translation parameters $T_x$ and $T_y$ are added to the origin. Similarly for the sphere, all the three translations mentioned above are added to the center to obtain the transformed position. This is because these geometries are generated about the origin (centered) and the circle does not have the $z$ coordinate while the sphere remains the same with any number of rotations due to symmetry. The general transformation is carried out based on the following

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.12}$$

$$R_y = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.13}$$

$$R_z = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.14}$$

$$T_r = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (4.15)$$

$$T = T_r * R_x * R_y * R_z \qquad (4.16)$$

Here, $\alpha$ , $\beta$ and $\gamma$ are the desired angles of rotation about the $x$ , $y$ and $z$ axes. This $4\times4$ homogenous transformation matrix $T$ is multiplied by each point in the data set as follows to obtain the transformed data and these coordinates are written into corresponding data files.

$$\begin{bmatrix} x^{'} \\ y^{'} \\ z^{'} \\ 1 \end{bmatrix} = T * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad (4.17)$$

Table 1 below summarizes the experimental parameters used for all the non-linear geometries and the noise. Noise added has a mean of zero and a standard deviation of 0.1. As mentioned earlier, the length of the cylinder does not matter but is specified for completeness of the geometry.

The data for all these non-linear geometries are generated such that they are centered about the origin. However, all the data are transformed to a different position in 3D space and then the analysis is carried out to determine the best fit parameters for the corresponding primitive.

Table 1: Summary of parameters and noise levels for non-linear geometries

| Geometry | Parameters | Standard Deviation of Noise |
|---|---|---|
| Circle | Radius = 10 | 0.1 |
| Sphere | Radius = 10 | 0.1 |
| Cylinder | Radius = 15 | 0.1 |
| | Length = 30 | |
| Cone | Base Radius =10 | 0.1 |
| | Apex semi-angle = $30^o$ | |
| Torus | Major Radius = 8 | 0.1 |
| | Minor Radius = 2 | |

## Implementation

This section presents the mathematical formulation of the least squares fitting problem for each analytic primitive and its implementation on the CPU as well as the graphics hardware (GPU). The algorithms and the programs written in C++ and CUDA are described. The GPU is programmed via the kernel to perform the compute intensive operations identified for all implementations while the C++ implementations are straight forward as explained. The benchmarking technique and the results for both forms of the implementations are discussed in the next chapter.

### Circle in a plane

A circle is defined by its center and radius. The equation of a circle is given by

$$x - x_0 \;^2 + \; y - y_0 \;^2 = r^2 \qquad (4.18)$$

Since any point on a circle must satisfy the above equation, it can be expanded, simplified and expressed as a linear system,

$$Ax = B \tag{4.19}$$

where $A$ is a matrix of dimension $n \times 3$ containing the $n$ data points with coordinates $(x_i, y_i)$ and a column of ones resulting from the linearization. $x$ is the design vector of dimension $3 \times 1$ and $B$ is the right hand side vector of dimension $n \times 1$.

Solving this over-determined system in the least squares sense, i.e.

$$A^T A x = A^T B \tag{4.20}$$

$$x = \left( A^T A \right)^{-1} A^T B \tag{4.21}$$

yields the design vector $x$ consisting of the coordinates of the center $(x_0, y_0)$ and the radius $r_0$. These values are used as initial estimates for the minimization procedure to find the best fit parameters. The error or the distance equation for a circle is given by equation (4.22)

$$f = \sqrt{(x_i - x_0)^2 + (y_i - y_0)^2} - r_0 \tag{4.22}$$

The objective function which is to be minimized is the sum of squares of the error which is given by

$$F(x_0, y_0, r_0) = \sum \left( \sqrt{(x_i - x_0)^2 + (y_i - y_0)^2} - r_0 \right)^2 \tag{4.23}$$

The expression (4.22) is evaluated at each data point and also the initial estimates of the parameters. This is assembled in a vector. The elements of the Jacobian matrix

$J$ are found from the partial derivatives of the distance equation $f$ with respect to each of the parameters $x_0, y_0, r_0$ which are given by

$$\frac{\partial f}{\partial x_0} = -(x_i - x_0) / \quad f + r_0 \tag{4.24}$$

$$\frac{\partial f}{\partial y_0} = -(y_i - y_0) / \quad f + r_0 \tag{4.25}$$

$$\frac{\partial f}{\partial r_0} = -1 \tag{4.26}$$

These partials are evaluated at each data point and initial guess to populate the Jacobian. This matrix and the vector of values for the objective function expressed as a linear system of equations,

$$Ju = -d \tag{4.27}$$

Where, $J$ is the Jacobian of dimension $n \times 3$, $u$ is vector of dimension $3 \times 1$ containing the parameter updates and $d$ is a vector of dimension $n \times 1$ whose elements are the distance function $f$ evaluated at each data point. The system of equations is over-determined and is solved as follows to obtain the updates

$$u = \quad J^T J \quad ^{-1} J^T \quad -d \tag{4.28}$$

The parameters are updated after each iteration as given by the equations below

$$x_0 = x_0 + u1 \tag{4.29}$$

$$y_0 = y_0 + u2 \tag{4.30}$$

$$r_0 = r_0 + u3 \tag{4.31}$$

Where $u1$, $u2$ and $u3$ are the elements of the vector $u$. These iterations are repeated until the change in parameters is negligible. The last updated parameters are the least squares best fit parameters for a circle.

C++ Implementation

The C++ implementation of the aforementioned procedure is described in this section. A class "Data" is created to model the coordinate data points. The class uses the vector template to store the data read in from the corresponding data file. It consists of a copy constructor, constructor, destructor, input operator and an overloaded output operator. It also provides full encapsulation and public interface to enable reading, writing and manipulation of the data. The class allows passing object references rather than the object itself.

Another class "Matrix" is created to perform the required mathematical computations on the data. This class also consists of a copy constructor, constructor and a destructor. Four essential functions are created, transpose () to multiply the transpose of a matrix with itself and return the product, jacobian () to populate the Jacobian matrix, chold() to perform the Cholesky decomposition to solve the matrix obtained by multiplying its transpose with itself and runTest() to iteratively solve for the vector containing the updates and add it to the parameters. Since the linear system of equations is over-determined, i.e., the number of rows of the matrix are greater than the number of columns, we obtain a symmetric positive definite matrix upon multiplication and hence the Cholesky decomposition is implemented.

The algorithm is as follows.

- The main() function reads the data from the file and stores in a “Data” class object.
- The object is passed into the constructor of the class “Matrix” by reference and the data are stored in an array.
- The array is passed by reference into the function transpose() to obtain the product and the right hand side vector which are then passed to chold() to obtain the initial estimates of the center and radius.
- This vector along with the array containing the data are passed into the function runTest() which performs the Gauss-Newton minimization.
- The functions jacobian(), transpose() and chold() are called sequentially within runTest() to obtain the updates.

The convergence condition is specified by declaring a tolerance value in the order of $10^{-3}$ and comparing the sum of the updates ($u1+u2+u3$) to this value. A condition is set such that the iterations stop when the sum is lower than the specified tolerance.

The implementation is tested for different sizes of data and is observed to provide accurate and reliable results.

### CUDA Implementation

The motivation of this research is to implement the existing minimization algorithms for the least squares fitting of analytic primitives on the GPU to explore its capabilities in parallelizing the computations involved thus cutting down the computation time. However, due to certain limitations such as the hardware and software model, and the inherent nature of the problem itself, it is necessary to identify specific parts of the computation that are to be delegated to the GPU rather than the entire computation.

The graphics card used for the implementation is NVIDIA Corporation's state-of-the-art GeForce 8800 GTX (G80) which is one of the superior graphics processors currently available in the market. The software interface Compute Unified Driver Architecture (CUDA) is used for issuing and managing computations on the GPU. This consists of a minimal set of extensions to the C programming language that allows targeting portions of the source code for execution on the card. The instruction set or code is compiled by a built in NVIDIA CUDA Compiler (NVCC). The program consists of two components,

- The host component that runs on the CPU (host) and provides functions to control and access the operations on the card.
- The device component also called the "kernel" which is described in the previous chapter. This component is executed on the GPU (device) in blocks consisting of threads.

The GPU has 3 memory partitions; the local or device memory, the constant memory and the shared memory. The data to be processed is copied to the device memory from the host. This is then passed onto the shared memory from where it is accessed by threads in each block to execute the instructions.

The source code contains two main functions:

- runTest(), a host function serving as wrapper to the kernel.

- kernel(), the device function (kernel) that executes the required computation on the device.

The arithmetic intensive operations involved in the least squares minimization algorithm for the circle fit include population of the Jacobian and the right hand side vector, multiplication of the transpose of this matrix with itself and the vector and finally, the Cholesky decomposition and forward substitution to solve the system of equations.

However, the memory model adopted to obtain maximum utilization of the computational power of the hardware does not permit implementation of all of the above mentioned mathematical operations. The computations executed on the GPU involve population of the Jacobian and the right hand side vector. The CUDA implementation of this procedure is similar to the C++ implementation explained in the previous section. For this implementation, the host or wrapper function runTest() is the one within which the Gauss Newton minimization procedure is performed. This function takes in data set and the vector of initial estimates as inputs performs the following operations:

- It allocates enough global memory to store the data array, the initial estimates and the results using cudaMalloc().
- It copies the coordinate data and the initial estimates from host memory to global memory using cudaMemcpy().
- The execution configuration parameters for the kernel function- the number of blocks and the number of threads in each block (block size) are set up. For the circle fit, the block size used is $16 \times 16$ and the number of blocks in the grid is selected such that it is a multiple of the block size and equal to $n$, the number of

points .The function kernel() is then executed to populate the Jacobian and the right hand side vector.

- The inputs to the function kernel() are the same as those for runTest() except that additional pointers for the results are passed in and these pointers point to the device memory instead of host memory.
- Within the function kernel() the following sequence of operations occur:
    a. $x$ and $y$ coordinate data for the circle are separately loaded as blocks on to the shared memory using the thread indices.
    b. The arithmetic instructions required to populate the Jacobian and the right hand side vector are specified separately for each set of coordinate data. The function syncthreads() is used to ensure that the data in all blocks are processed.
    c. Finally, the processed data are copied back to the device memory such that each array or vector forms a separate column of the Jacobian and the last array copied back to the device forms the right hand side vector.
- The results are copied back from the device to the host (CPU) using the function cudaMemcpy() with each column of the Jacobian as well as the right hand side vector being stored in separate arrays.
- The arrays containing the columns of the Jacobian are assembled into a single array and this is passed into the function transpose() along with the right hand side vector. This function returns the product of the transpose of the Jacobian with itself and the right hand side vector.

- These are passed into the function chold() which computes the Cholesky decomposition of the $3 \times 3$ product matrix and solves the resulting system by forward substitution to obtain the updates for the parameters.
- The parameters are then updated and the entire procedure is iteratively repeated until the convergence condition is satisfied. The updated parameters form the new estimates and are copied back on to the GPU at the beginning of each iteration.

The convergence condition and the tolerance value for the termination of this iterative procedure remains the same as that in the C++ implementation. The source codes from the two implementations are tested for different data sets and the results are found to be accurate.

### Sphere

The sphere is analogous to a circle. It is parameterized by in 3D space by its center ( $x_0, y_0, z_0$ ) and radius $r_0$ .The equation is given by

$$x - x_0 \quad ^2 + \quad y - y_0 \quad ^2 + (z - z_0)^2 = r_0{}^2 \qquad (4.32)$$

Since the only parameter different in a sphere from a circle is the z-coordinate, the above equation can be linearized to form the linear system of equations in equation (4.19) where the matrix A is of size $n \times 4$ and the design vector $x$ is of size $4 \times 1$. This system when solved as mentioned earlier yields the design vector $x$ which consists of the initial estimates of the coordinates of the center $x_0, y_0, z_0$ and the radius $r_0$ .The distance function or the error for the sphere is given by

$$f = \sqrt{\;x_i - x_0\;{}^2 + \;y_i - y_0\;{}^2 + \;z_i - z_0\;{}^2} - r_0 \tag{4.33}$$

The Gauss-Newton method is used to minimize the objective function for a sphere which is given by

$$F\;\; x_0, y_0, z_0, r_0 \;\; = \sum \;\sqrt{\;x_i - x_0\;{}^2 + \;y_i - y_0\;{}^2 + \;z_i - z_0\;{}^2} - r_0\;\;{}^2 \tag{4.34}$$

The partial derivatives of the function $f$ with respect to each of the parameters $x_0, y_0, z_0, r_0$ are given by

$$\frac{\partial f}{\partial x_0} = -(x_i - x_0) / \;\; f + r_0 \tag{4.35}$$

$$\frac{\partial f}{\partial y_0} = -(y_i - y_0) / \;\; f + r_0 \tag{4.36}$$

$$\frac{\partial f}{\partial z_0} = -(z_i - z_0) / \;\; f + r_0 \tag{4.37}$$

$$\frac{\partial f}{\partial r_0} = -1 \tag{4.38}$$

The Jacobian matrix and right hand side vectors are populated in the same manner as that for the circle. However, the size of the Jacobian matrix is now $n \times 4$ and the right hand side vector is $4 \times 1$. This is again solved using the relation given by equation (4.28) to obtain the updates which in this case is a $4 \times 1$ vector.

The parameters are updated after each iteration

$$x_0 = x_0 + u1 \tag{4.39}$$

$$y_0 = y_0 + u2 \tag{4.40}$$

$$z_0 = z_0 + u3 \tag{4.41}$$

$$r_0 = r_0 + u4 \tag{4.42}$$

These iterations are repeated until the algorithm converges and the last updated parameters are the least squares best fit parameters for a sphere.

C++ Implementation

The C++ implementation for the finding the least squares best fit parameters of a sphere is very similar to that for a circle. The class “Data” mentioned in the previous section is modified to read in the z-coordinate from the file created in MATLAB. The class “Matrix” also mentioned in the previous section is modified to store and manipulate the coordinate points. Necessary changes are made in the functions transpose(), jacobian() and runTest()to account for the change in the size of the array and vectors. The size of the Jacobian in this case is $n \times 4$, the right hand side remains $n \times 1$ and the vector of updates has a size $4 \times 1$. The algorithm for this implementation is also the same as described in the previous section.

The sum of the updates $(u1+u2+u3+u4)$ is compared to the tolerance value (same as that for circle) to check for convergence and the iterations are terminated when the sum goes below the tolerance. The implementation has been tested for different sizes of coordinate data sets and the tests indicate that the results are accurate and reliable.

CUDA Implementation

Since the solution procedure for the circle and the sphere is the same, the parts of the computation implemented on the GPU are valid for the sphere as well. The host and device source codes for the CUDA implementation used earlier (runTest() and kernel()

respectively) remain the same as that for a circle. The two functions take in the same inputs and return similar elements as outputs. The algorithm and the sequence of operations is also the same as that explained in the CUDA implementation of the circle. The only significant differences between the two implementations are

- The specification of the size for the allocation of memory on the host and the device for the vectors and arrays used is changed to accommodate for the three dimensional aspect or the $z$ -coordinate of the sphere.
- The function kernel() takes in an additional pointer as input to store and return the additional column in the Jacobian.

The convergence condition for the termination of the minimization loop within the function runTest() and the tolerance value remain the same. Execution of the program with different data sets indicates that the implementation is accurate.

**Plane**

The plane is a linear geometry which is defined by a point $x, y, z$ and the direction cosines of the normal to it $a, b, c$ . The problem of finding the least squares best fit plane is a rather simple Eigen-value problem. The distance from any point $x_i, y_i, z_i$ in space to a plane is given by

$$d = a.\ x_i - x\ + b.\ y_i - y\ + c.\ z_i - z \qquad (4.43)$$

The sum of the squares of this error forms the objective function to be minimized. This is given by

$$F\ \ x, y, z, a, b, c\ = \sum\ a.\ x_i - x\ + b.\ y_i - y\ + c.\ z_i - z\ \ ^2 \qquad (4.44)$$

It can be shown that the least-squares plane passes through the centroid or mean of the data $\bar{x}, \bar{y}, \bar{z}$ [20]. Hence the centroid of the coordinate data can be considered as the point that defines the plane. It is required to find the direction cosines associated with this point. It is shown that the Eigen vector corresponding to the smallest Eigen value of the matrix formed by subtracting the mean from each point in the data set is the normal to the least squares plane. The matrix is formulated as

$$A = \; x_i - \bar{x}, y_i - \bar{y}, z_i - \bar{z} \qquad (4.45)$$

Where

$$\bar{x} = \frac{\sum x_i}{n} \qquad (4.46)$$

$$\bar{y} = \frac{\sum y_i}{n} \qquad (4.47)$$

$$\bar{z} = \frac{\sum z_i}{n} \qquad (4.48)$$

Typically, a Singular Value Decomposition of the matrix given by (4.45) is done and the vector corresponding to the smallest singular value of the matrix $A$ forms the direction cosines of the normal to the plane. This is analogous to finding the Eigen values of $A^T A$ which is a matrix of size $3 \times 3$. In our case, for simplicity of implementation, the Eigen values are computed by finding the roots of the cubic equation obtained by setting up the determinant of the $3 \times 3$ matrix. Therefore, the centroid of the data and the Eigen vector corresponding to the minimum Eigen value of the above mentioned matrix define the least squares best fit plane.

C++ Implementation

Since the plane is a linear geometry, the solution involved is not iterative. A C++ source code is written to perform the sequence of operations explained above. The algorithm for this implementation is

- The execution starts with the main function which again uses the class “Data” to read in the $x, y, z$ coordinate data representing a plane and stores it in a vector.
- The function mean() takes in the vector as input and returns the centroid of the data ($\bar{x}, \bar{y}, \bar{z}$).
- The data along with the vector containing the mean is then passed into the function create_matrix() which returns the matrix whose columns are formulated by subtracting the centroid of the data from the $x$, $y$ and $z$ coordinates.
- The matrix is passed in as an argument to the function transpose() which returns the $3 \times 3$ product matrix.
- The function cal_abc() then computes the coefficients $a$, $b$ and $c$ of the cubic equation.
- The function cubic_root() is coded to find the roots of a cubic equation. The coefficients computed in the previous step are passed into this function which returns the roots or Eigen values of the matrix.
- These are then compared to determine the minimum value which is passed into the function eig_vec() where the corresponding Eigen vector is computed based on Cramer’s rule.

Thus, the best fit plane to three dimensional coordinate data are defined by the point which is the centroid of the data and the Eigen vector computed above which represents the direction cosines of the normal to the plane.

CUDA implementation

The compute intensive operations involved in the least squares fit procedure of a plane are matrix multiplication, calculation of cubic roots and the computation of the Eigen vector. Ideally, parallelizing all these operations would result in a significant gain in computation time but due to certain limitations of the hardware, and also with some experimentation, only the portion of the computation involving the translation of the data by the centroid is implemented on the GPU.

The source code for the GPU implementation is the same as the C++ implementation. The major difference is that the function create_matrix() from the C++ implementation is modified such that it forms the wrapper for the function kernel() which executes the data translation operation on the GPU. The inputs to this function include 3D coordinate data for the plane and the vector containing the centroid. In this case it is modified such that it returns the product $A^T A$ where $A$ is the matrix obtained by translating the data by the centroid and is populated on the GPU. The function transpose() to compute the product matrix is called from within the function create_matrix().

The execution configuration parameters – the block size and the grid size are the same for all implementations. The number of threads in a block is $16 \times 16$. i.e., there are 256 threads in all blocks and the grid size or the number of blocks in a grid is computed based on the size of the data set. In general, the number of blocks are chosen such that the number is a multiple of 256 and the total number of threads in all blocks is equal to the total number of elements in the data set. The data along with the centroid is copied to the device memory. The $x, y$ and $z$ coordinates are loaded onto shared memory as separate blocks and suitable instructions are specified to execute the data translation operation. The translated data are copied back to the host memory as separate vectors and assembled into one single dimensional array. The sequence of operations that follows is the same as that explained in the algorithm for the C++ implementation. Furthermore, the functions in the source code used to execute the rest of the computation involved are the same as those used in the C++ implementation.

**Utility functions**

The distances from a point in space to a line and to plane or the line and plane distance functions are an essential part of the next three non-linear geometries – cylinder, cone and torus. These are therefore defined as given in [20] . The distance from a point $x_i, y_i, z_i$ to line defined by a point $x_0, y_0, z_0$ and normalized direction cosines $a,b,c$ is given by

$$f_i = \sqrt{u^2 + v^2 + w^2} \tag{4.49}$$

Where

$$u = c * \; y_i - y_0 \; - b * \; z_i - z_0 \tag{4.50}$$

$$v = a * \; z_i - z_0 \; - c * \; x_i - x_0 \tag{4.51}$$

$$w = b * \; x_i - x_0 \; - a * \; y_i - y_0 \tag{4.52}$$

The distance from a point $x_i, y_i, z_i$ to a plane defined by a point $x_0, y_0, z_0$ and the normal direction $a,b,c$ is given by

$$g_i = a* \;\; x_i - x_0 \;\; + b* \;\; y_i - y_0 \;\; + c*(z_i - z_0) \qquad (4.53)$$

The derivatives for the expressions (4.49) and (4.53) with respect to certain defining parameters play an important role in the non-linear least squares minimization of the aforementioned geometries and are given in [20]

**Cylinder**

A cylinder is defined by a point $\;\; x_0, y_0, z_0 \;\;$ on its axis, a vector $\;\; a, b, c \;\;$ pointing along the axis (the direction cosines) and its radius $r$ .The distance from a point to the surface of a cylinder is given by

$$d = f_i - r \qquad (4.54)$$

Where $f_i$ is the distance from a point in space to a line given by equation (4.49) and $r$ is the radius of the cylinder. The objective function is given by

$$F \;\; x_0, y_0, z_0, a, b, c, r \;\; = \sum \;\; f_i - r \;\; ^2 \qquad (4.55)$$

The initial estimates for the defining parameters of a cylinder are obtained by considering the ideal case that all points lay on the surface. In other words, the error is considered to be zero. By equating the relation given by (4.54) to zero, rearranging the terms and squaring the resulting expression yields

$$u^2 + v^2 + w^2 = r^2 \qquad (4.56)$$

Substituting for $u$ , $v$ and $w$ in the above equation and simplifying gives

$$Ax_i^{\;2} + By_i^{\;2} + Cz_i^{\;2} + Dx_iy_i + Ex_iz_i + Fy_iz_i + Gx_i + Hy_i + Iz_i + J = 0 \qquad (4.57)$$

The coefficients $A$ to $J$ are given by

$$A = (b^2 + c^2) \tag{4.58}$$

$$B = (a^2 + c^2) \tag{4.59}$$

$$C = (a^2 + b^2) \tag{4.60}$$

$$D = -2ab \tag{4.61}$$

$$E = -2ac \tag{4.62}$$

$$F = -2bc \tag{4.63}$$

$$G = -2(b^2 + c^2)x_0 + 2aby_0 + 2acz_0 \tag{4.64}$$

$$H = 2abx_0 - 2(a^2 + c^2)y_0 + 2bcz_0 \tag{4.65}$$

$$I = 2acx_0 + 2bcy_0 - 2(a^2 + b^2)z_0 \tag{4.66}$$

$$J = (b^2 + c^2){x_0}^2 + (a^2 + c^2){y_0}^2 + (a^2 + b^2){z_0}^2 - 2bcy_0z_0 - 2acx_0z_0 - 2abx_0y_0 - r^2 \tag{4.67}$$

Equation (4.57) is normalized with respect to its first coefficient $A$ and expressed as a system of linear equations. This is solved in the least squares sense to obtain a vector consisting of the remaining nine normalized coefficients $\frac{B}{A}, \frac{C}{A}, \frac{D}{A}, \frac{E}{A}, \frac{F}{A}, \frac{G}{A}, \frac{H}{A}, \frac{I}{A}$ and $\frac{J}{A}$. These elements are compared for their proximity to 1 and 0 and based on the relations in equations (4.59) through (4.63) the initial estimates for the direction cosines $(a, b, c)$ are computed. Once the direction cosines are determined, the definitions of the coefficients $G$, $H$, $I$ and the relation between the direction cosines and the axis

$$ax_0 + by_0 + cz_0 = 0 \tag{4.68}$$

a system of linear equations in the form of equation (4.19) is set up and solved in the least squares sense to obtain the initial estimates of the point on the axis $x_0, y_0, z_0$ . The initial estimate of the radius is then computed by substituting the values computed into equation(4.67).

The Gauss-Newton minimization method is modified to find the least squares best fit parameters for a cylinder. At the beginning of each iteration a copy of the cylinder data are translated so that the point on the axis is the origin of the coordinate system. The data are then rotated to be aligned with the $z$ -axis (standard position) by multiplying the data with a suitable rotation matrix. The advantage of this constraint is that it reduces the time to evaluate the derivatives of the distance function. This in turn simplifies the computation involved in setting up and solving the Jacobian. The partial derivatives of the distance function for a cylinder in standard position are given by

$$\frac{\partial d}{\partial x_0} = -x_i / f_i \tag{4.69}$$

$$\frac{\partial d}{\partial y_0} = -y_i / f_i \tag{4.70}$$

$$\frac{\partial d}{\partial z_0} = 0 \tag{4.71}$$

$$\frac{\partial d}{\partial a} = -x_i * z_i / f_i \tag{4.72}$$

$$\frac{\partial d}{\partial c} = -y_i * z_i / f_i \tag{4.73}$$

$$\frac{\partial d}{\partial c} = 0 \tag{4.74}$$

$$\frac{\partial d}{\partial r} = -1 \tag{4.75}$$

In the general case, the Jacobian would be an $n \times 7$ matrix of the partial derivatives of the distance function with respect to the seven defining parameters. By applying the technique mentioned, the point on the axis is at $0,0,0$ and the direction cosines are constrained to be $0,0,1$ . The derivatives then reduce to the equations (4.69) through(4.75). Consequently, evaluating these partials at each data point is significantly simplified and the Jacobian also reduces to an $n \times 5$ matrix which is easy to solve. The right hand side vector is again the distance function evaluated at all data points. The Jacobian and the right hand side vector are solved to obtain the updates. Since these updates are for the cylinder in standard position, the parameters are updated in a different manner to obtain the values in the original position. If the rotation matrix is defined by $U$ and the vector of updates is defined by $p$ which consists of 5 elements, then the parameters are updated as follows:

$$\begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + U^T * \begin{pmatrix} p_1 \\ p_2 \\ -p_1 * p_3 - p_2 * p_4 \end{pmatrix} \tag{4.76}$$

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = U^T * \begin{pmatrix} p_3 \\ p_4 \\ 0 \end{pmatrix} \tag{4.77}$$

$$r = r + p_5 \tag{4.78}$$

Where, $U^T$ is the transpose of the rotation matrix, $p_1$ through $p_5$ are the values of the updates for $x_0$, $y_0$, $a$, $b$ and the radius $r$ in the same order. It can be seen the rotation matrix does not affect the radius of the cylinder as it remains a constant. This procedure is performed iteratively until the solution has converged. In the first step of the subsequent iteration the copy of the original data set is used rather than the transformed data set from the previous iteration. The tolerance for the convergence specified is $10^{-3}$. The tolerance value is greater in this case as the algorithm is sensitive to noisy data. In this case, the sum of the updates $p_1$ through $p_5$ is compared to the tolerance value to determine whether the solution has converged. The other condition is the number of iterations which is set to a hundred. In other words, the iterations stop when the value of the sum of updates is less than the tolerance value or when the number of iterations is hundred.

C++ Implementation

The entire source code for the C++ implementation is written in a modular fashion, i.e., different functions are defined and written to carry out each step of the procedure explained above. The structure of the source code is similar to the circle and sphere implementations. The algorithm for the procedure is

- The main() uses the class “Data” to read in coordinate data from the corresponding file and stores it in a vector template.
- This is passed into a function calc_AJ() wherein the coefficients of equation (4.57) normalized with respect to the first coefficient are computed and returned in a vector.
- The function then calc_abc() takes in this vector as input and returns the initial estimates of the direction cosines.
- The direction cosines are then passed into the function calc_xyz_r_0() which returns the initial estimates of the point on the axis and the radius.

- The initial estimates along with the coordinate data are then passed into the function runTest() where the iterations for the minimization are carried out. The sequence of operations within this function are:
    a. The data and initial estimates are first copied into different variables.
    b. A call to the function calc_rot_mat_vec() is made to compute the rotation matrix required to rotate the coordinate data from its current position to the standard position $0,0,1$. This function is written in a generic manner to return the suitable rotation matrix required to rotate data whose axis is defined by one direction vector to another direction vector.
    c. The rotation matrix, the copy of the data and initial estimates are passed in as inputs to the function transform() which first translates the data and then rotates it so that the data are in the standard position.
    d. The functions jacobian(), transpose() and chold() are then called in the same order to set up the Jacobian and right hand side vector, multiply the transpose of the Jacobian with itself and the right hand side vector, and finally to solve the system to obtain the updates respectively.
    e. The parameters are updated as explained and the entire procedure is repeated until the convergence condition is met.
- The function runTest() then returns the last updated values of the parameters which form the solution.

CUDA implementation

Apart from two arithmetic intensive operations; the population of the Jacobian and the right hand side vector which were the identified in the circle and sphere fits, the transformation of the data, i.e., the translation and rotation of the 3D data to be axis aligned at the beginning of every iteration is also implemented on the GPU in the case of the cylinder fit. The source code again consists of two major components namely runTest()-host component and kernel()-the device component. The initial estimates are computed within the main() function by using the necessary functions explained in the C++ implementation. These along with the coordinate data are inputs to the function runTest(). The sequence of operations within runTest() remain the same except that the operations executed in the functions transform() and Jacobian() are executed on the GPU. Since a copy of the original data are transformed to be axis aligned at the beginning of every iteration, the data transfer between the host and the device is avoided by loading the data onto the device once and storing it in an array on the device memory for use in every subsequent iteration.

The structures of the host and device components are the same as that of the CUDA implementations for the circle and sphere. The major difference is in the inputs to the kernel function. In the case of the cylinder, the rotation matrix and the point on the axis of the cylinder which are required for the data transformation are the additional inputs. Furthermore, the kernel function also takes additional pointers to device memory as inputs to store the columns of the Jacobian matrix which in the case of the cylinder is five. The instructions within the function kernel() are also suitably changed to perform the transformation on the data and operate on the transformed data to obtain the Jacobian and right hand side vector according to the equations presented earlier. The results are copied back to the host and passed into the function transpose() as inputs. The resukting product matrix and vector obtained are solved within the function chold() to obtain the updates. The parameters are then updated as explained in the C++ implementation and the updated parameters are used to generate the new rotation matrix and the entire procedure is repeated until the solution has converged and the last updated parameters are the best fit parameters. The convergence condition and tolerance remain the same as explained earlier.

**Cone**

The defining parameters for a cone are a point on its axis $x_0, y_0, z_0$ which is not the apex, the direction cosines of the axis $a, b, c$ , the orthogonal distance from the point on the axis to the cone $s$ and its apex semi-angle denoted by $\psi$ . The distance equation or the error for a cone is given by

$$d = f_i * \cos\psi + g_i * \sin\psi - s \qquad (4.79)$$

Where $f_i$ is the distance from a point $x_i, y_i, z_i$ in space to a line which in this case is the axis of the cone given by equation(4.49). $g_i$ is the distance from a point to a plane defined in equation (4.53) and $s$ is defined above. The objective function is the sum of the squares of this error given by

$$F(x_0, y_0, z_0, a, b, c, s, \psi) = \sum \ f_i * \cos\psi + g_i * \sin\psi - s^{\ 2} \qquad (4.80)$$

Since the distance equation cannot be linearized, the initial estimates for the parameters in the case of the cone are guesses. The concept of finding the least squares best fit cone to 3D coordinate data are the same as that for the cylinder. A copy of the data are transformed using the rotation matrix at the beginning of each iteration such that it is in standard position or aligned with the $z$ -axis. This technique simplifies the computation of partial derivatives of the distance equation with respect to the defining parameters as the point on the axis and direction cosines are known in the standard position. The partial derivatives of (4.79)are dependent on the partial derivatives of the line and plane distance functions $f_i$ and $g_i$ respectively. For the general case, the partials for the distance equation of the cone in simplified form are

$$\frac{\partial d}{\partial x_0} = -x_i * \cos\psi / f_i \tag{4.81}$$

$$\frac{\partial d}{\partial y_0} = -y_i * \cos\psi / f_i \tag{4.82}$$

$$\frac{\partial d}{\partial z_0} = 0 \tag{4.83}$$

$$\frac{\partial d}{\partial a} = -x_i * z_i * \cos\psi / f_i + y_i * \sin\psi \tag{4.84}$$

$$\frac{\partial d}{\partial b} = -y_i * z_i * \cos\psi / f_i + y_i * \sin\psi \tag{4.85}$$

$$\frac{\partial d}{\partial c} = 0 \tag{4.86}$$

$$\frac{\partial d}{\partial s} = -1 \tag{4.87}$$

$$\frac{\partial d}{\partial \psi} = -f_i * \sin\psi + g_i * \cos\psi \tag{4.88}$$

However, in the event that a point or points in the data set lies on the axis (generally the apex), the line distance function $f_i$ is zero. Hence its partial derivatives with respect to the point on the line and the direction cosines are not defined as explained in [20]. This causes the partials for the distance equation of the cone also to be undefined at these points. The gradient for line distance function is modified for such cases and hence the partial derivatives of (4.79) with respect to $x_0$, $y_0$, $a$ and $b$ are given by

$$\frac{\partial d}{\partial x_0} = \cos\psi \tag{4.89}$$

$$\frac{\partial d}{\partial y_0} = \cos\psi \tag{4.90}$$

$$\frac{\partial d}{\partial a} = z_i * \cos\psi + x_i * \sin\psi \tag{4.91}$$

$$\frac{\partial d}{\partial b} = z_i * \cos\psi + y_i * \sin\psi \tag{4.92}$$

The remaining derivatives remain the same. The transformed data and initial estimates are used to compute the Jacobian matrix as well as the right hand side vector. The Jacobian matrix is now an $n \times 6$ matrix instead of an $n \times 8$ matrix. The line and plane distance functions, $f_i$ and $g_i$ are computed using the inputs. The Jacobian is then populated by using equations (4.89) through (4.92) when the value of $f_i$ is zero and equations (4.83) through (4.88) otherwise. The rest of the solution procedure remains the same as that for the cylinder. There are six updates for the parameters in this case The point on the axis and the direction cosines for the cone are updated in the same manner as given by equations (4.76) and(4.77). The remaining two values in the vector containing the updates are added to the orthogonal distance $s$ and the apex semi-angle $\psi$ respectively as these do not change immaterial of the cone's orientation The iterations are terminated either when the sum of updates is below the tolerance specified ($10^{-3}$) or when the number of iterations is hundred.

C++ Implementation

Two separate programs are developed to treat the two possible cases for the cone. The first case is theoretical wherein the data set is assumed to have a point or a set of points sampled off the apex of the cone during the process of measurement. In the second case, which occurs in reality, the coordinate data sampled from a surface of the cone is assumed to have no apex. This is a more common case because in reality the apex is rounded off and hence no points are sampled off the surface during measurement. Although the programs for both the aforementioned cases are essentially the same, the major difference between them is discussed in this section.

The C++ implementation of the least squares fit for a cone is a variation of that for the cylinder. The functions implemented to compute the initial estimates are eliminated and the function jacobian() is modified to accommodate the necessary changes in computation which explained in this section. The user entered initial estimates and data read in from the file using the "Data" class are passed into the function runTest() which returns the best fit parameters. The sequence of operations within this function is the same as that for the cylinder.

The transformed data and estimates are inputs to the function jacobian(). The line and plane distance functions are evaluated at each data point. For the case with apex singularity, an *if* condition is set to determine if the value of line distance function $f_i$ is zero at any data point. If this condition is true, the corresponding elements of the Jacobian matrix are computed by using relevant equations [(4.89)-(4.92)] and if it is false the equations (4.81) through (4.88) are used. The right hand side vector is the distance function itself evaluated at each data point in the point cloud. The Jacobian which is an array of size $n \times 6$ and the vector of size $n \times 1$ are passed into the function transpose() which returns the product of the transpose of the Jacobian with itself and the right hand side vector. The $6 \times 6$ product matrix and the $6 \times 1$ vector are solved by Cholesky decomposition and forward substitution to obtain the updates. The parameters are updated and the iterations are terminated according to the convergence conditions as explained in the previous section.

The only significant difference in the C++ implementation for the second case (cone data with no apex) is within the function jacobian() wherein the control flow instruction *if* is eliminated and the elements of the Jacobian are computed in a straight forward manner using equations (4.81) through (4.88).

The drawback of the implementation is that it requires good initial guesses of the parameters as the algorithm is not numerically robust. The number of iterations required for the solution to converge increases with poor initial guesses.

### CUDA Implementation

Similar to the C++ implementations, two separate GPU (CUDA) implementations are developed for the two cases mentioned in the least squares fit of a cone. The structure of the implementation remains the same as that for cylinder. The operations executed on the GPU include transformation of the data to be axis aligned, population of the Jacobian matrix and the right hand side vector. The function runTest() forms the platform for the kernel() function which executed theses operations. Memory allocation for the input and the results on the device id carried out using cudaMalloc(). The coordinate data are loaded on to the device in a one time data transfer using cudaMemcpy(). This is stored in an array on device memory and used for the transformation at the beginning of every iteration in the minimization procedure. Thus, unnecessary data transfer between the device and host is avoided lending the implementation efficient. The functions transform() and jacobian() from the C++ implementation are replaced by kernel() to which executes the mentioned computations on the GPU. The block and grid sizes for the execution of the kernel are specified in the same manner as explained for all previous implementations. The inputs to the function kernel() are different from that for the cylinder implementation. In this case, a pointer to the array required to store another column of the Jacobian matrix is one additional input.  For the case where the apex singularity of the cone is taken into consideration, an *if* condition is specified within the kernel to determine at which points the value of the line distance function is zero and the Jacobian is populated using the necessary mathematical equations as explained previously. However, this control flow instruction is specified such that all the threads in a warp and in general within all the blocks follow the same path of the condition to avoid serializing the operations. The performance of this implementation is compared to the one in which the apex singularity for the cone is neglected. In this case, the control flow instruction is eliminated and the elements of the Jacobian are computed in the same manner as explained in the C++ implementation. The outputs from the kernel are copied back to the host (CPU) and solved to obtain the updates. The initial estimates are then updated as explained previously and the new estimates are used to generate the new rotation matrix. The elements of this matrix along with the updated estimates are copied back to the device and the procedure is repeated until the solution converges.

**Torus**

The torus is a non-linear geometry which is parameterized by its 3D center $x_0, y_0, z_0$ , the direction cosines of its axis $a,b,c$ , the major radius $r$ and the minor radius $R$ . The distance from any point $x_i, y_i, z_i$ to the surface of the torus is given by

$$d=\sqrt{g_i^{2}+\ \ f_i-r\ ^{2}}-R \tag{4.93}$$

Where $f_i$ and $g_i$ are the line and plane distance functions respectively which are defined previously The sum of the squares of this distance from each point in the data set to the torus is the objective function to be minimized. This is given by

$$F(x_0, y_0, z_0, a,b,c,r,R)=\sum\ \ \sqrt{g_i^{2}+\ \ f_i-r\ ^{2}}-R\ ^{2} \tag{4.94}$$

The distance function given by equation cannot be linearized and hence initial estimates have to be good guesses. The strategy used to solve for the best fit parameters for the cylinder and the cone is used for the torus as well. The data are transformed to the standard position at the beginning of every iteration of the Gauss Newton minimization procedure. Since the data are in the standard procedure, the partial derivatives of the distance equation with respect to the parameters are simplified and given by

$$\frac{\partial d}{\partial x_0}=\frac{-x_i*\ 1-r/f_i}{d+R} \tag{4.95}$$

$$\frac{\partial d}{\partial y_0}=\frac{-y_i*\ 1-r/f_i}{d+R} \tag{4.96}$$

$$\frac{\partial d}{\partial z_0}=\frac{-z_i}{d+R} \tag{4.97}$$

$$\frac{\partial d}{\partial a}=\frac{x_i*z_i*\ r/f_i}{d+R} \tag{4.98}$$

$$\frac{\partial d}{\partial b} = \frac{y_i * z_i * \;\; r / f_i}{d + R} \tag{4.99}$$

$$\frac{\partial d}{\partial c} = 0 \tag{4.100}$$

$$\frac{\partial d}{\partial r} = \frac{- \;\; f_i - r}{d + R} \tag{4.101}$$

$$\frac{\partial d}{\partial R} = -1 \tag{4.102}$$

These partial derivatives are evaluated at each point of the coordinate data and the initial estimates and assembled in columns to form the Jacobian matrix which in this case is of $n \times 7$, $n$ being the size of the data set. The right hand side vector is the distance function evaluated at all the data points and is of size $n \times 1$. These are then solved in the least squares sense using the previously explained procedure to obtain the vector of updates which in this case is a $7 \times 1$ vector ( $p$ ). The first three updates $p_1, p_2, p_3$ are for the point on the axis, the next two $p_4, p_5$ are for the direction cosines and last two values $p_6, p_7$ of the vector are for the major and minor radius. The procedure to update the parameters remains the same as that for the cone and the cylinder except for the center point of the torus which is updated as follows:

$$\begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + U^T * \begin{pmatrix} p_1 \\ p_2 \\ (-p_1 * p_4 - p_2 * p_5) * p_3 \end{pmatrix} \tag{4.103}$$

Where $U^T$ is the transpose of the rotation matrix required to rotate the data to the standard position. The noticeable change here is the inclusion of the update $p_3$ for the $z$ coordinate of the center point. The tolerance value for the convergence remains $10^{-3}$ but condition for convergence is set such that the iterations terminate when the value of sum of first five updates is compared to tolerance or the limit for the number of iterations (hundred) is reached.

### C++ Implementation

The structure and sequence of operations in the C++ implementation for the torus fit remains the same as that for the cone. All functions that were used for the various steps in computations in this implementation are reused for the torus. The only significant changes made are in the function jacobian() where the formulae used to populate and store the Jacobian matrix and the right hand side vector are changed. Consequently, the other change is in the size of array used to store the Jacobian matrix. The product of the transpose of the Jacobian with itself is a $7 \times 7$ matrix which is solved to obtained the updates using the function chold(). The update for the center point of the torus is programmed according to equation (4.103) while the rest of the updates are similar to those in the implementation for the cone. The iterations are terminated according to the convergence conditions specified in the previous section.

### CUDA Implementation

Since the entire procedure, algorithm and the computations involved for finding the least squares best fit torus to data are very similar to the cone, the graphics hardware implementation for this primitive is also developed on the same lines. The only significant changes are

- In the size of memory allocated on the device and the number of variables used to store the intermediate results and outputs. The kernel program in this case takes in an additional input which is a pointer to the array used to store the additional column in the Jacobian.

- The updates for the center point of the torus is changed according to equation (4.103).

The rest of the source code remains the same as that for the cone. The last updates parameters are the least squares best fit parameters for a torus.

# CHAPTER V

# RESULTS AND DISCUSSIONS

This chapter describes the procedure developed to benchmark parts of computation carried out on the GPU against those on the CPU. The CPU used for the C++ implementation is a 2.13 GHz Intel Core 2 Duo processor with a 2.0 GB Random Access Memory (RAM) while the GPU as is an NVIDIA GeForce 8800 GTX card with an on board memory of 768MB. The implementations are suitably time profiled and the results obtained are presented with relevant observations.

## **Benchmarking**

The C++ and graphics hardware (CUDA) implementations of the least squares of all non-linear geometries viz. circle, sphere, cylinder, cone and torus are similar in structure and are benchmarked against each other to determine the gain in computation time. Since the computations for which the time comparison is to be drawn are within the Gauss-Newton minimization loop, a procedure for timing these specific operations in both C++ as well as the GPU implementations is developed and is explained in this section. The method is valid for all the aforementioned non linear geometries. However the functions timed in the case of the circle and sphere are different from those for the cylinder, cone and torus and this is discussed in this section.

In the GPU implementations for the circle and sphere fits, the operations executed on the GPU include setting up the Jacobian and right hand side vector. These operations are carried out in a separate function called jacobian() on the CPU (C++ implementation). Hence, in both cases a variable is declared to record the time in each iteration and the times for the subsequent iterations are added to get the total computation time. The standard Win32 function QueryPerformanceCounter() is used to time the operation in the case of the CPU implementations. This function returns the current value of the high-resolution performance counter. The timing function is invoked before the function jacobian() and terminated just after its execution. The device or GPU part of the code is timed by using CUDA timer functions viz. cutCreateTimer(), cutStartTimer(), cutStopTimer() and cutGetTimerValue(). In the first iteration, the value returned by these functions is stored in a variable. The timer is reset at the start of the next iteration and the value returned is added to the value previously stored. Thus the total computation time is obtained. The functions to reset and start the timer are invoked before copying the updated parameters from the host device in the case of the circle, sphere, cylinder, cone and torus. The call to terminate the timer is made after the results are copied back from the device to host. However, the functions timed in the CPU implementations for the cylinder, cone and torus are the transform() and jacobian() as the operations executed on the GPU include transformation of the data as well as population of the Jacobian and the right hand side vector.

The benchmarking technique for either implementations of the plane fit is straight forward as it does not involve iterative solution. The only operation executed on the GPU involves translation of the coordinate data by its centroid. The timer functions for the GPU implementation are invoked before the data and vector containing the centroid is loaded on to the device memory and terminated after the resulting matrix has been copied back to the host memory for further computation. Whereas for the CPU implementation, the function create_matrix() is timed as the translation of the data are executed within this function.

## Results

All the CPU and GPU implementations are executed with data sets of sizes ranging from 8000 to about 12 million data points to establish a deterministic relation between the way the CPU implementations scale with large data streaming as compared to the way GPU implementations perform. Table 2 includes the computation times for the circle and sphere fits. The times taken for the execution of operations in the cylinder and torus fits are tabulated in Table 3. Table 4 gives a comparison of the theoretical and practical cases for a cone fit. Finally, Table 5 shows the computation times for the CPU and GPU implementations of the plane fit.

Table 2: Results for Circle and Sphere fit

| **CPU Vs GPU** | | | | |
|---|---|---|---|---|
| **Time (msecs)** | | | | |
| | **Circle** | | **Sphere** | |
| Data size | CPU | CUDA (GPU) | CPU | CUDA (GPU) |
| 8192 | 0.578 | 0.237 | 1.109 | 0.626 |
| 16384 | 1.005 | 0.392 | 2.254 | 0.938 |
| 36864 | 2.265 | 0.779 | 3.978 | 1.966 |
| 65536 | 4.866 | 1.251 | 5.758 | 2.994 |
| 98304 | 7.097 | 1.863 | 7.985 | 4.357 |
| 147456 | 10.128 | 3.318 | 17.43 | 6.245 |
| 196608 | 13.145 | 3.874 | 22.297 | 9.023 |
| 262144 | 17.245 | 4.687 | 32.423 | 11.62 |
| 393216 | 25.608 | 6.91 | 59.117 | 16.574 |
| 589824 | 37.412 | 10.317 | 88.392 | 25.761 |
| 786432 | 57.483 | 13.945 | 117.719 | 32.158 |
| 1048576 | 124.608 | 34.913 | 157.229 | 48.396 |

Table 3: Results for Cylinder and Torus fit

| **CPU Vs GPU** | | | | |
|---|---|---|---|---|
| **Time (msecs)** | | | | |
| | **Cylinder** | | **Torus** | |
| Data size | CPU | CUDA (GPU) | CPU | CUDA (GPU) |
| 8192 | 8192 | 2.498 | 14.02 | 2.544 |
| 16384 | 16384 | 4.972 | 21.709 | 3.461 |
| 36864 | 36864 | 11.203 | 51.821 | 6.137 |
| 65536 | 65536 | 20.075 | 96.942 | 9.813 |
| 98304 | 98304 | 31.078 | 153.129 | 13.963 |
| 147456 | 147456 | 49.517 | 263.684 | 20.295 |
| 196608 | 196608 | 66.095 | 333.897 | 27.361 |
| 262144 | 262144 | 87.712 | 472.126 | 36.771 |
| 393216 | 393216 | 131.267 | 663.213 | 51.096 |
| 589824 | 589824 | 197.464 | 1042.09 | 83.055 |
| 786432 | 786432 | 262.523 | 1461.5 | 109.672 |
| 1048576 | 1048576 | 349.01 | 1927.7 | 138.984 |

Table 4: Results for Cone fit

| **CPU Vs GPU** | | | | |
|---|---|---|---|---|
| **Time (msecs)** | | | | |
| | **Cone (with Apex)** | | **Cone (without Apex)** | |
| Data size | CPU | CUDA (GPU) | CPU | CUDA (GPU) |
| 8192 | 11.001 | 2.504 | 10.85 | 2.448 |
| 16384 | 21.618 | 4.134 | 19.898 | 4.066 |
| 36864 | 51.095 | 7.911 | 48.569 | 7.217 |
| 65536 | 96.215 | 12.7 | 91.007 | 11.784 |
| 98304 | 162.154 | 18.6005 | 158.081 | 17.642 |
| 147456 | 255.563 | 26.754 | 252.86 | 26.393 |
| 196608 | 331.55 | 34.969 | 328.208 | 34.151 |
| 262144 | 463.381 | 46.281 | 461.094 | 44.329 |
| 393216 | 681.066 | 68.663 | 675.849 | 67.859 |
| 589824 | 1037.34 | 101.545 | 1031.01 | 100.571 |
| 786432 | 1381.08 | 138.341 | 1372.04 | 136.389 |
| 1048576 | 1833.96 | 179.692 | 1824.06 | 178.345 |

Table 5: Results for Plane fit

| CPU Vs GPU | | | |
|---|---|---|---|
| | Time (msecs) | | |
| **Data size** | **CPU** | **CUDA(GPU) with data transfer** | **CUDA (GPU) without data transfer** |
| 16384 | 0.068 | 0.535 | 0.113 |
| 36864 | 0.184 | 1.057 | 0.207 |
| 65536 | 0.531 | 1.817 | 0.563 |
| 98304 | 0.668 | 2.56 | 0.811 |
| 147456 | 0.867 | 3.752 | 0.974 |
| 196608 | 1.15 | 4.97 | 1.217 |
| 262144 | 1.662 | 7.037 | 1.446 |
| 393216 | 2.752 | 9.277 | 2.574 |
| 589824 | 4.837 | 14.869 | 4.682 |
| 786432 | 6.147 | 20.13 | 6.031 |
| 1048576 | 8.67 | 26.65 | 8.61 |
| 1572864 | 13.077 | 38.772 | 12.652 |

It is observed from the tables that the GPU implementations are significantly faster than the CPU implementations in all cases except the plane fit. The tremendous gain in computational speed for these cases is mainly due to the memory model adopted. In other words, the manner in which the data are loaded on to the shared memory and operated on contributes significantly to the performance gain. As explained earlier, the $x$, $y$ and $z$ coordinates for any primitive are processed independently of each other as individual threads in a block. Many such blocks are processed simultaneously in a grid and thus any arithmetic instruction specified for a particular data set is executed simultaneously on all elements in the set which results in the performance boost.

The plane fit on the other hand is a closed form solution and the only operation carried out on the GPU is the translation of the data. The entire data are copied on to the device for processing. The overhead of copying the data back and forth from the device to the host is greater than any gain in computation time. Furthermore, since this operation involves subtraction of a single value from a set of blocks, all the threads from different blocks tend to access the same memory location causing bank conflicts. This serializes all the computation and hence no significant gain is observed. All the results tabulated above are for data sets consisting of a million points. However, as mentioned above the programs are executed with data sets of up to 12 million points to gain a better insight on the performance of the hardware as the data sizes scale up. A statistical analysis of the data is carried out to determine the relation between the computation time and the size of data set for all primitives.

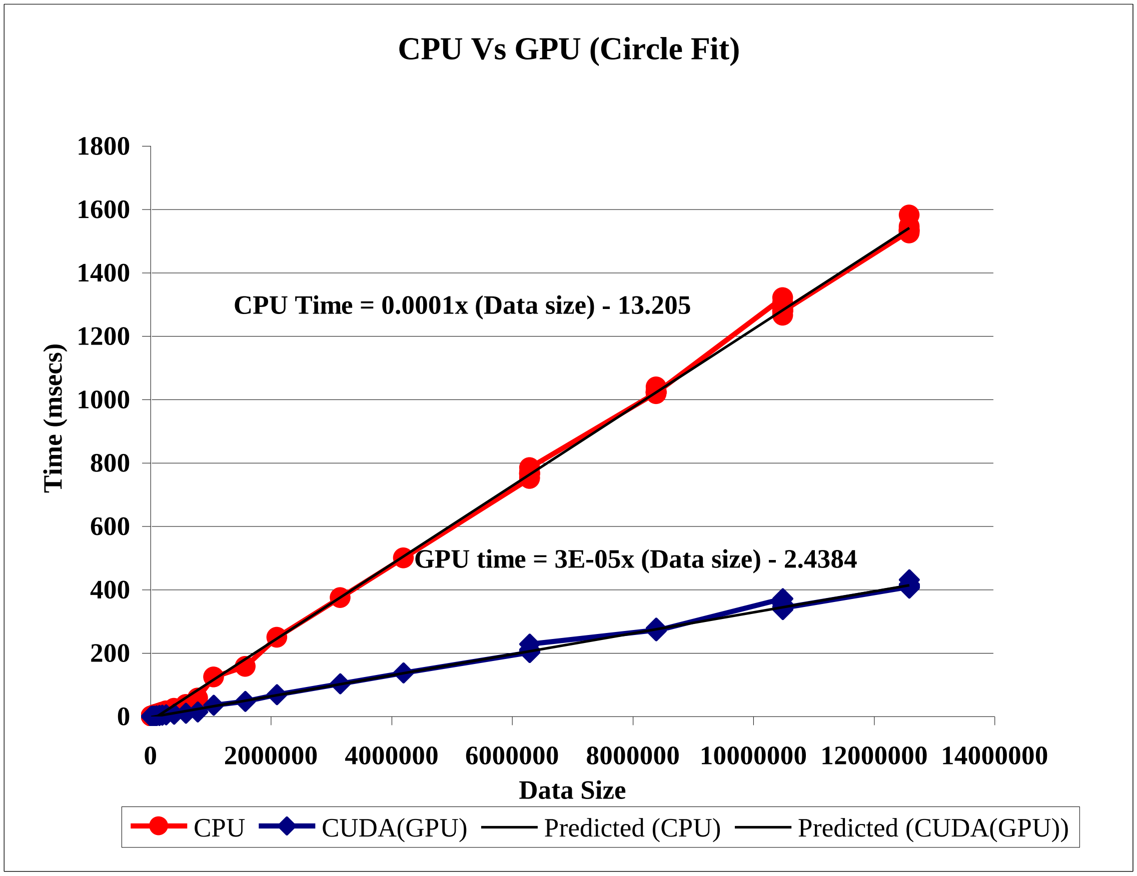

Figure 10: CPU Vs GPU (Circle Fit)

The comparison of computation times of the CPU and GPU for a circle fit for data sets consisting of approximately 8000 points to about 12 million points is plotted in Figure 10 based on the results tabulated in Table 2. A linear regression is carried out on the data to determine the relation between the times taken for increase in data set sizes. The observation is that for very large data sizes, the computation time on the GPU is about 0.3 times the computation of the CPU owing to the parallel processing capabilities of the hardware. In other words the GPU is about 3-4 times faster than the CPU in performing the same computations. This is determined by taking the ratio of the slopes between the two trend lines from the regression analysis as shown in Figure 10. It is noticed that the trend line for the CPU Vs Data size has a negative y-intercept. This does not give the exact interpretation of the process as it is not practically possible that the CPU takes negative time when there are no data points. However, this trend can be attributed to the fact that for some data sets, the increase in time taken does not follow a linear trend causing the line to have a negative y-intercept. This is because of the randomly generated noise added to the data. Some data sets take lesser computation times than the predicted value by the linear model while others take more than the predicted value. The trend line for the GPU results on the other hand has a positive y-intercept which accounts for the set up time required within the kernel program.

To ensure that the noise in the data is the cause for the trends observed, four data sizes are chosen and ten data sets for each of these data sizes are generated with the same noise level. These are used as inputs to execute the CPU and GPU implementations. The error bars corresponding to these data sets shows the effect of noise. The y-intercept is negative for the CPU trend line because of the high computation times for certain data sets especially the ones consisting of a large number of points.

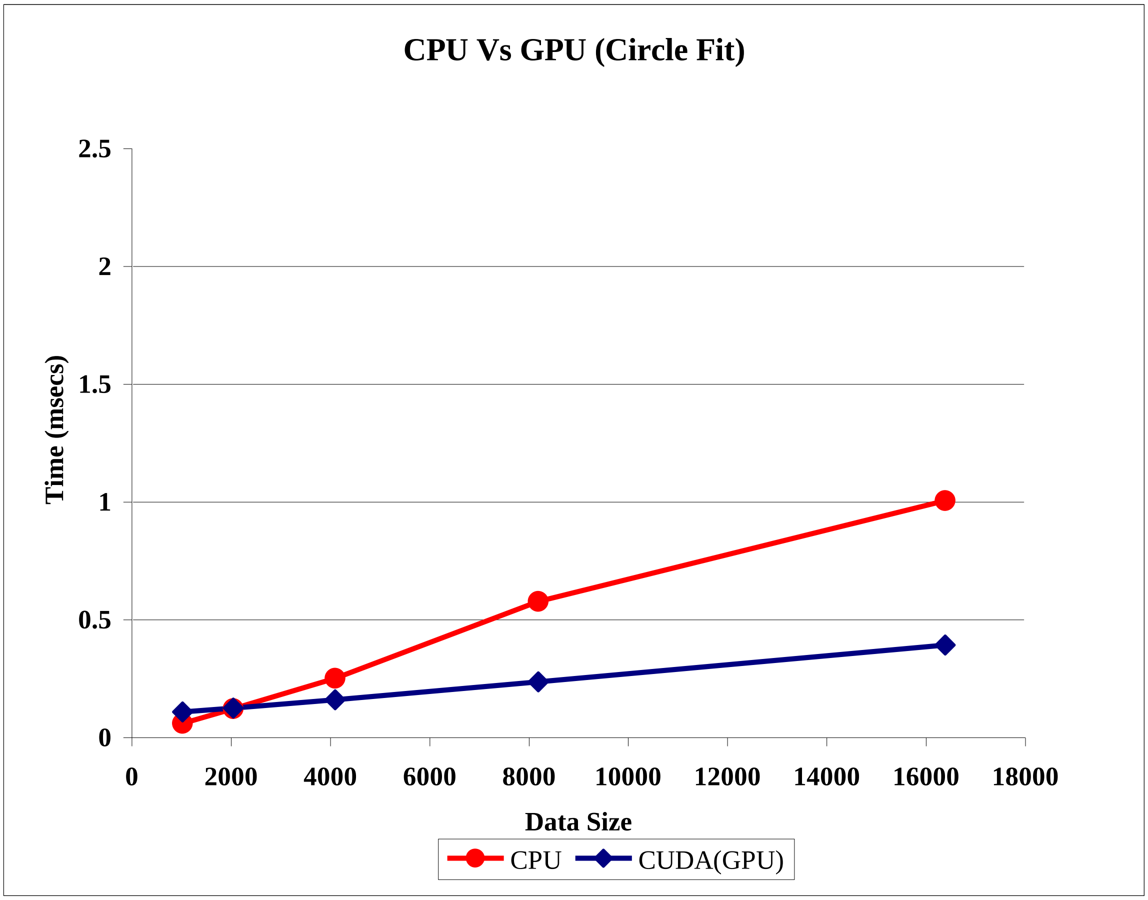

Figure 11: CPU Vs GPU (Circle Fit with small data sets)

Another interesting aspect in this implementation is the number of points beyond which the GPU out performs the CPU. The memory model adopted requires that a minimum of 1024 points be processed to obtain the results. Hence, the implementations are benchmarked with smaller data sets than earlier and these are plotted above in Figure 11. It is evident that the CPU outperforms the GPU when the number of points in the data set is less than 4096. This indicates that when the computational complexity is less, which is the case with the circle fit, the GPU is good with data sets which contain more than 4000 points.

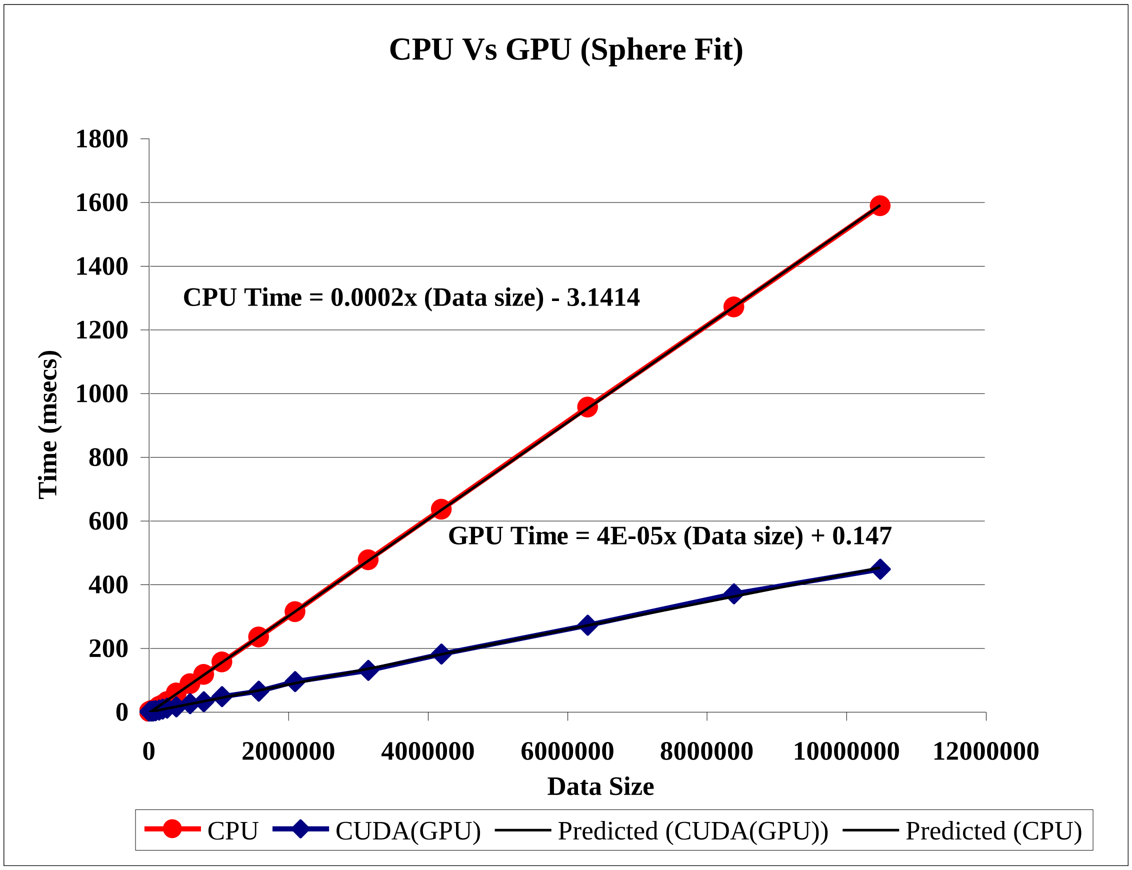

Figure 12: CPU Vs GPU (Sphere Fit)

A similar analysis is carried out on the time data for the sphere fit in Figure 12 which is based on the values tabulated in Table 2. However, the maximum size of the data set in this case is about 10 million. This is because the number of variables required is greater which induces a memory limitation during execution. The ratio of the slopes of the trend lines for the results from the two implementations gives us an estimate that on an average the GPU is about 5 times faster which implies that the total computation time for the operations on the GPU take about 0.2 times the total computation time on the

CPU. Furthermore, the trend line for the CPU results has a negative y-intercept which again indicates that the time taken for larger data sets by the CPU increases non-linearly. Experimentation with smaller data sets indicates that the GPU is slower than the CPU for the least possible number of points (1024). To determine the size of the data set for which the GPU outperforms the GPU the computation times for both implementations for small data sets are shown in Figure 13. In this case, as expected, the GPU outperforms the CPU for a data set consisting of 2048 points which is lesser than that for the circle fit (4096). There is an increase in computational complexity but the execution of all these operations in parallel on the GPU results in the performance gain even with a comparatively small data set. Hence, the minimum number of points for which the GPU can be used for computations for a sphere fit is 2048.

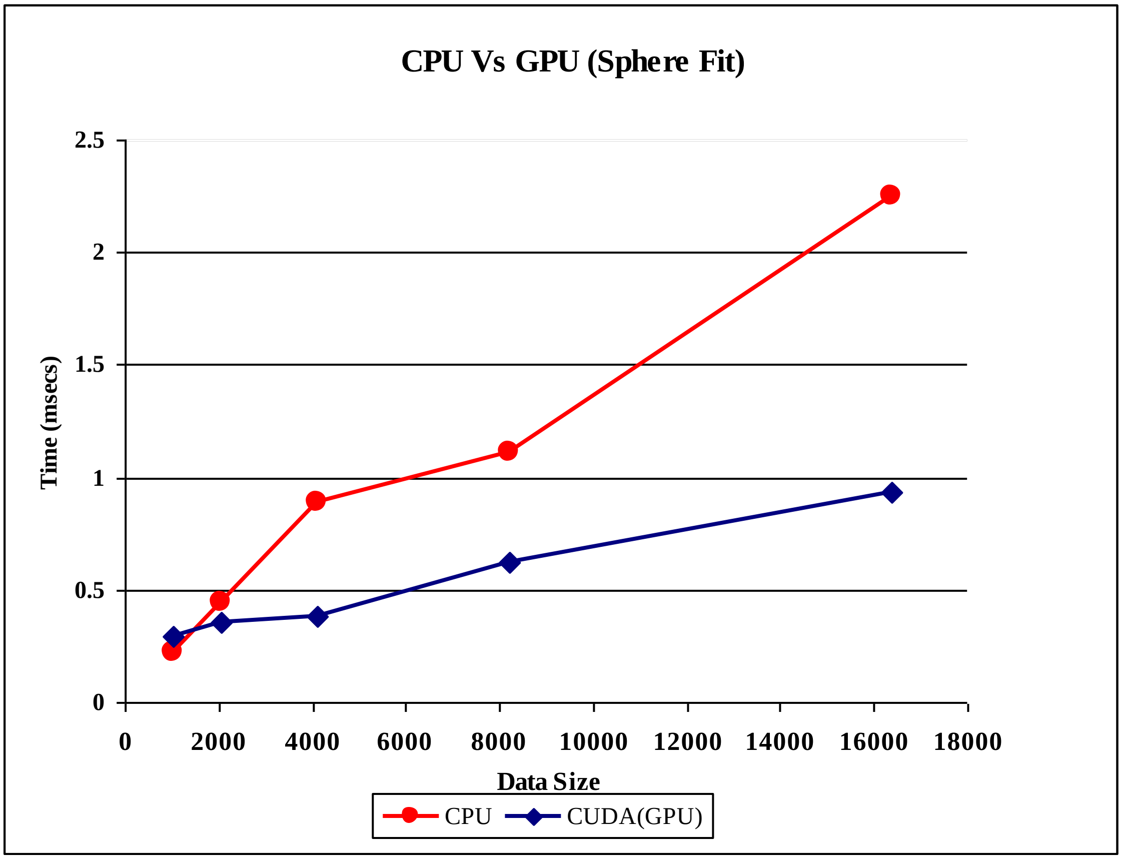

Figure 13: CPU Vs GPU (Sphere Fit with smaller data sets)

Figure 14 shows interpretation of the results from the two implementations of the cylinder fit which is based on the values tabulated in Table 3. As the number of variables required for computations is larger, the maximum size of the data set with which these implementations can be tested gets smaller and hence the number of points used in this analysis is limited to about 8 million.

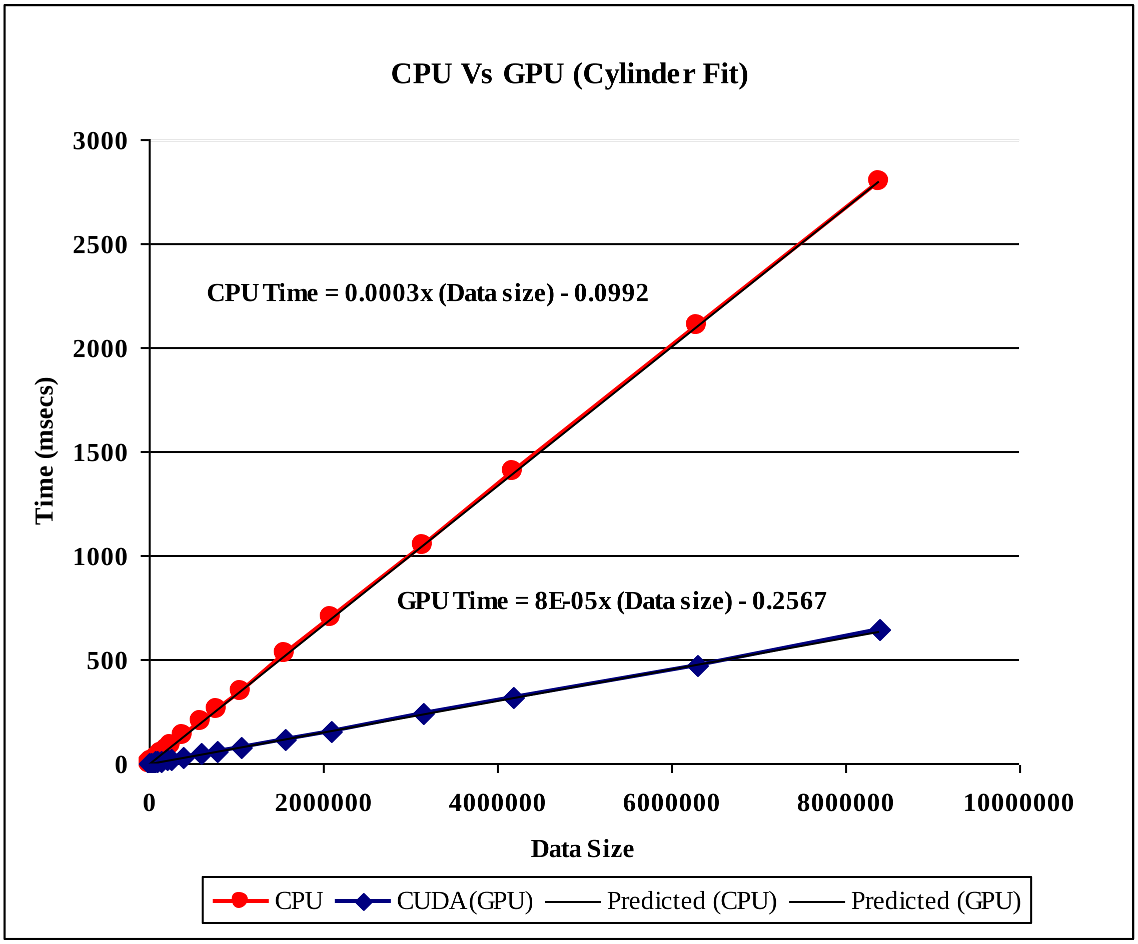

Figure 14: CPU Vs GPU (Cylinder Fit)

The regression analysis of the results obtained indicates that the GPU is about 3.75 times faster than the CPU for the data sets tested. This is determined by taking the ratio of the slopes from the trend lines obtained from the regression. This can also be expressed by stating that the computation on the GPU takes about 0.26 times the total computation time on the CPU. Although these trends contradict the trends observed in the circle and sphere fits which indicated that the increase in complexity increases the performance gain, there are two reasons for the observations made for the cylinder. First, the data set consists of about 8 million points as compared to the 10 million points used in the sphere and circle analysis. As seen earlier, the actual time taken for computations on the CPU does not increase linearly with very large data sets which would result in the slope of the CPU trend line being greater. Secondly, the computations performed on the GPU for the cylinder fit include rotation and translation of the data along with the population of the Jacobian and the right hand side vector. Since the rotation and translation operations involve various threads accessing the same memory location on shared memory, serialization of operations occur. However, the gain obtained in the population of the Jacobian and the right hand side vector overcomes this overhead.

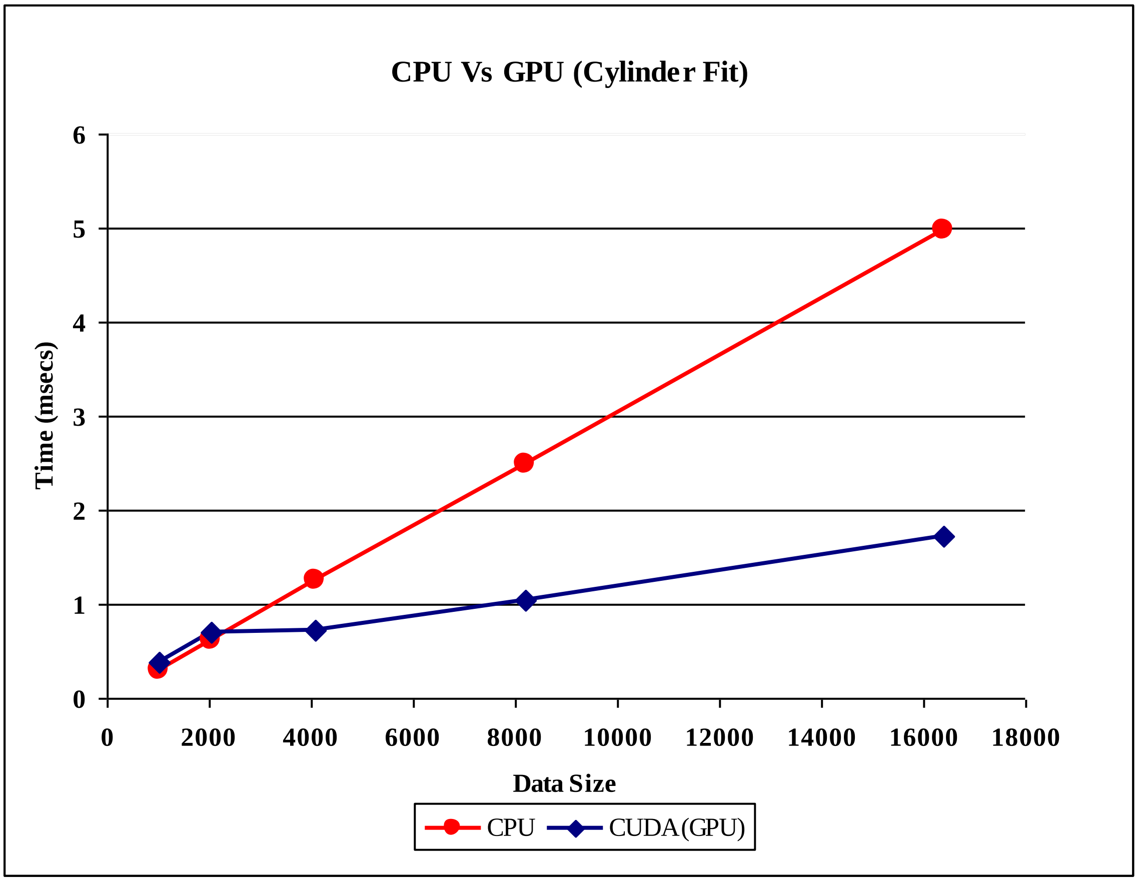

Figure 15: CPU Vs GPU (Cylinder Fit with smaller data sets)

Another important observation made from this analysis is that the GPU times for the last two data sets increases non-linearly resulting in the negative intercept of the trend line. Furthermore, the time profiling of both implementations for the minimum data size of 1024 data points reveals the CPU is faster than the CPU. Hence the implementations are tested with smaller data sets to determine the minimum number of points at which the GPU's performance gain is observed. This is depicted in Figure 15 and as observed the gain in performance of the GPU is for data sets consisting of over 4096 points.

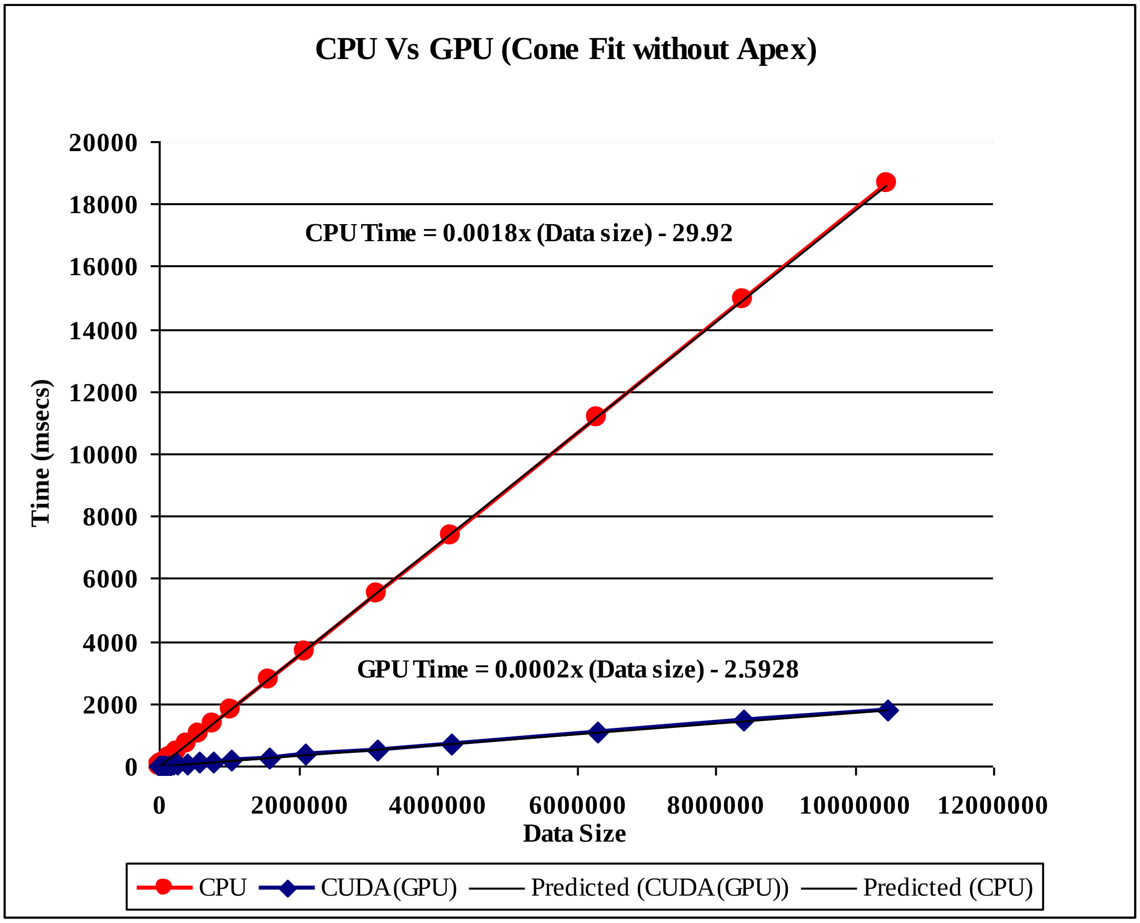

Figure 16: CPU Vs GPU (Cone Fit without Apex)

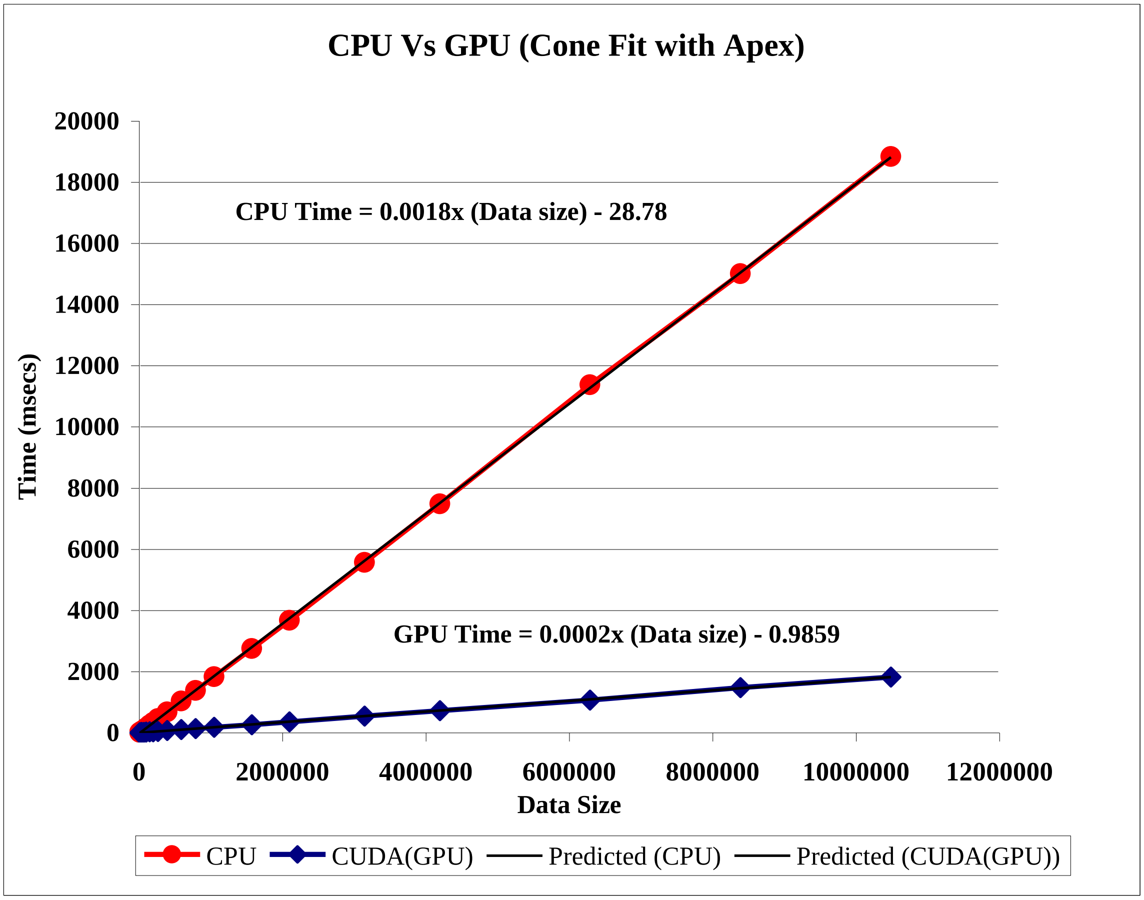

Figure 17: CPU Vs GPU (Cone Fit with Apex)

The results from the two separate cases for the least squares fit of a cone are analyzed. Figure 16 projects the practical case in which the data set does not contain points sampled off the apex of the cone. Figure 17 presents the theoretical case which includes the apex in the data. Both the figures are plotted based on the values recorded in Table 4. The analysis focuses on determining the effect of using the control flow instruction (*if*) on the instruction throughput or the efficiency of the GPU implementation. However, as seen from above, there is no significant change in computation time even with the inclusion of this statement although there is some difference. The control flow instruction for the GPU implementation is specified in such a manner that all the threads follow the same path of execution thus avoiding serialization of operations.

From both figures it is evident that the GPU takes about 0.11 of the time taken by the CPU to execute the same instructions. In other words, the GPU is approximately 9 times faster than the CPU in operation (ratio of slopes). This implies that the linear model used gives us a fairly good estimate as the actual data indicates that the number of times the GPU is faster than the CPU ranges from about 5-10. Further, the trend lines for the CPU and GPU in both cases have negative intercepts with the ones for the CPU being highly negative. This is again due to the noise in the data.

Unlike the trends observed in the cylinder fit the performance gain of the GPU for the cone fit is more pronounced though the same operations are executed in both cases. However, in the cone fit, the population of the Jacobian is more computationally complex than that for the cylinder and the parallelization of these operations on the GPU accounts for the gain.

Time profiling the CPU and GPU implementations for both cases of the cone fit for very small data set (1024 coordinate points) indicates that the GPU is about 1.3 times faster than the CPU for the cone with an apex while it is 1.4 times faster.

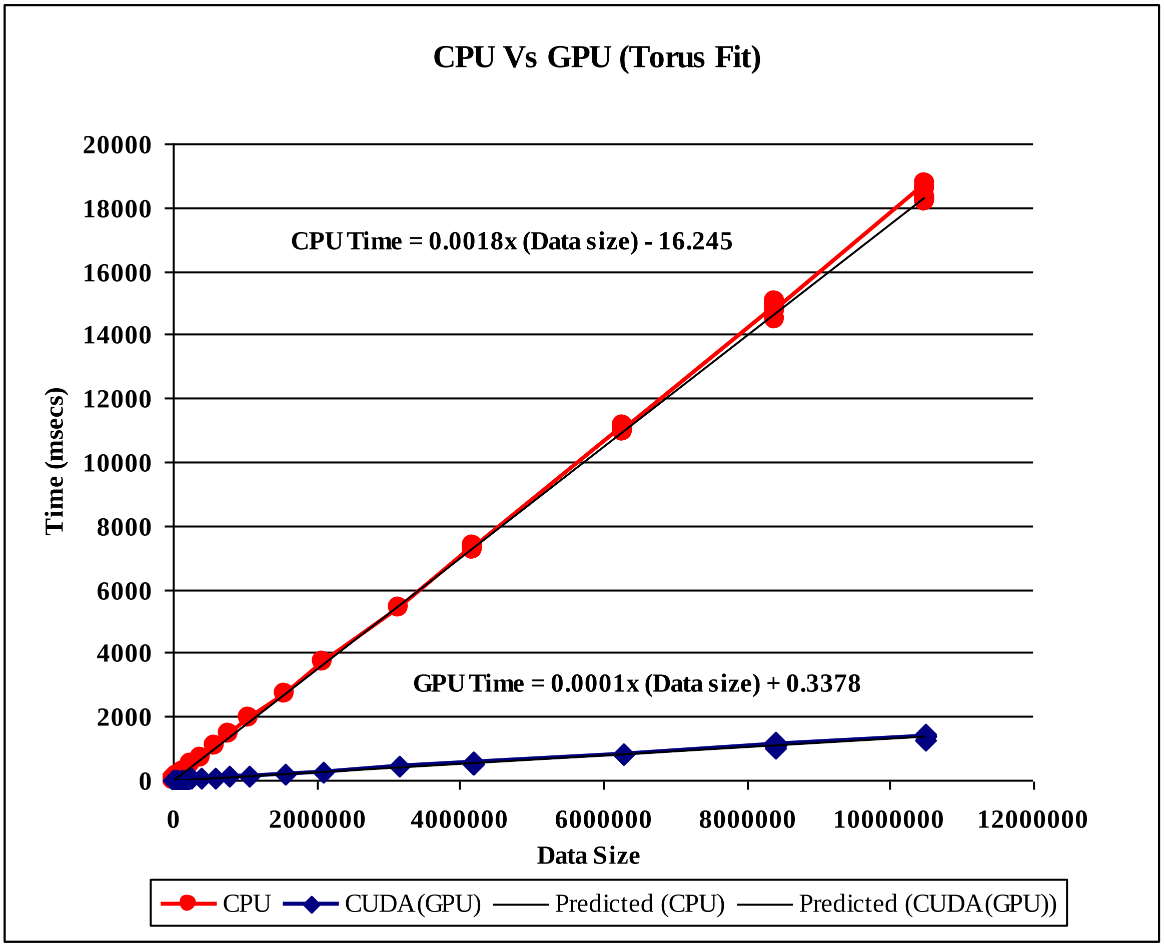

Figure 18: CPU Vs GPU (Torus Fit)

The results from the time profiling of the CPU and GPU implementations for the torus fit (Table 3) and the linear regression analysis of the same is presented in Figure 18. The GPU implementation in this case is again orders of magnitude faster than that of the CPU. The total time taken to execute the operations involving transformation of data and population of the Jacobian matrix and right hand side vector on the GPU is about 0.055 times the total time taken by the CPU to execute the same operations. The ratio of the slope of the CPU trend line to that of the GPU trend line indicates that the GPU is about 18 times faster than the CPU. The actual data indicates that the gain in performance is about 14. The effect of noise is more pronounced in this case which is evident with the high negative value of the y-intercept in the CPU trend line and the high value of the slope. As in the case of the circle fit, four data sizes are chosen and ten data sets for each of these are generated, and the implementations are executed. It is observed that the variations are slightly larger for the torus as compared to those in the case of the circle fit. This is because we have noise in 3D in the case of the torus, while the noise gets added only to the $x$ and $y$ coordinates for the circle. Furthermore, the GPU's performance gain is observed even with very small data sets which conform to the expectation that the GPU's performance increases significantly with increase in arithmetic intensity of operations being executed by it.

Further, experiments are conducted with the same data sets for the torus but at three different noise levels to determine the trends for the two implementations. In the first case, all the data are generated with a noise of zero while for the second case they are generated with a noise of standard deviation 0.01. The third case considered here presents the computation times for both implementations of the torus fit when the noise added is much higher (standard deviation ($\sigma$)of 0.5). Figure 19, Figure 20 and Figure 21 illustrate the analysis with the above mentioned data sets respectively.

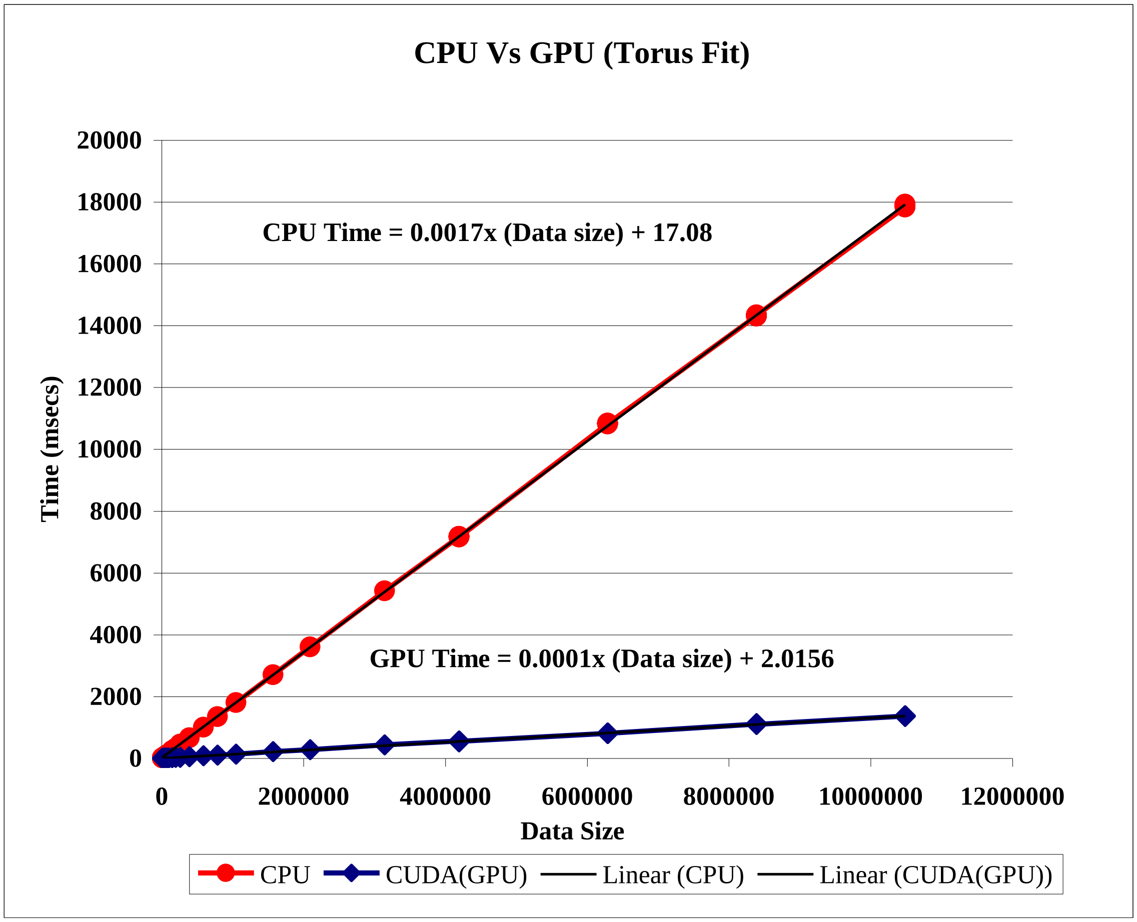

Figure 19: CPU Vs GPU (Torus Fit with no noise)

When the noise in the data is eliminated, the variation in computation times for both implementations is more linear as observed above. It is observed that the y-intercept is positive for the CPU trend line in this case. Further, the ratio of the slopes of the trend lines is 17 which conform to the expectation that the gap between the two lines should reduce in the absence of noise. The implementations are executed 10 times for each of the last four data sets. It is observed there is almost negligible change in computation times for the CPU and the GPU with each subsequent data set signifying the role of noise.

In the second case, as expected low noise (0.01) does not affect the computations times too much although there is some variation. This accounts for the negative y-intercept for the CPU trend line. Four largest data sizes are chosen and the two implementations are executed for 10 different data sets for each data size to determine the effect of noise. Figure 20 indicates the time spread for the last four data sets. The ratio of slopes for the two trend lines remains the same as that in Figure 19.

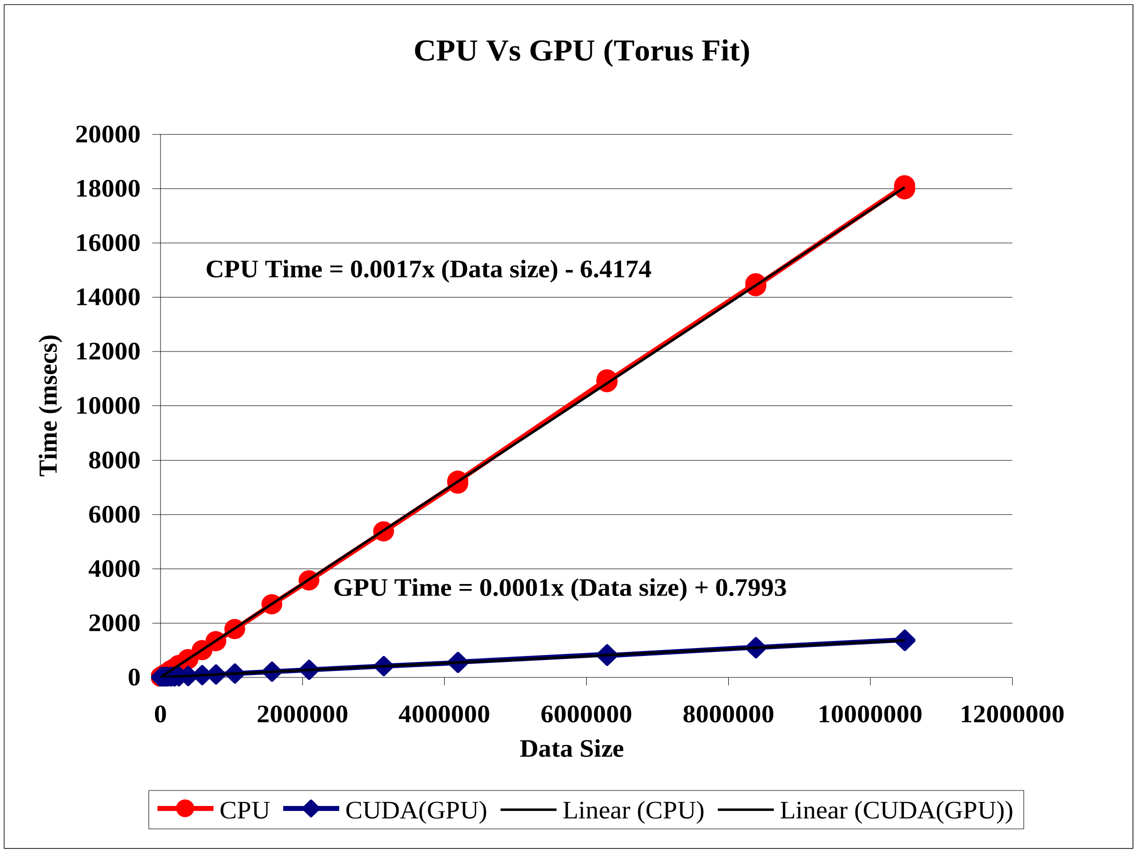

Figure 20: CPU Vs GPU (Torus Fit with low noise (0.01))

Magnifying the noise significantly increases the randomness in the data as seen in Figure 21. As explained previously, 10 data sets are generated for each of the last four data sizes are used for analysis. Although the both the trend lines have a positive y-intercept, their slopes vary significantly as compared to those in Figure 18. Furthermore, in this case the randomness is more significant for the data sets consisting of 8192 and

16384 points due to the effect of noise. This accounts for the high positive y-intercept values in the trend lines.

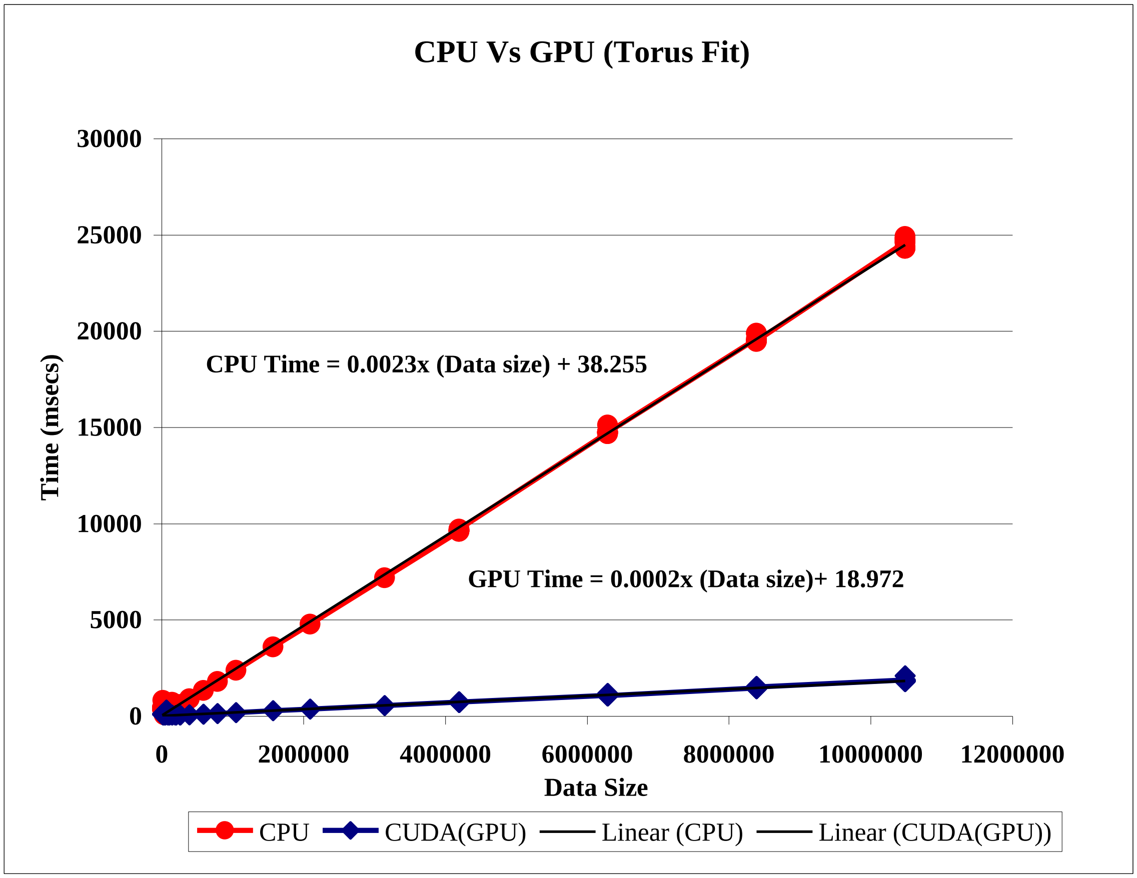

Figure 21: CPU Vs GPU (Torus Fit with high noise (0.5))

Table 6: Time spreads for various noise levels (Torus Fit)

| Number of Points | Time spread for Noise $\sigma$ = 0 (msecs) | | Time spread for Noise $\sigma$ = 0.01 (msecs) | | Time spread for Noise $\sigma$ = 0.1 (msecs) | | Time spread for Noise $\sigma$ = 0.5 (msecs) | |
|---|---|---|---|---|---|---|---|---|
| | CPU | GPU | CPU | GPU | CPU | GPU | CPU | GPU |
| 4194304 | 15.7 | 17.3 | 104.25 | 20.65 | 139.1 | 36.38 | 111.67 | 34.88 |
| 6291456 | 24.4 | 18.7 | 106.9 | 49.97 | 183.9 | 51.82 | 394.3 | 126.47 |
| 8388608 | 28.9 | 27.4 | 92.1 | 33.4 | 574.8 | 189.75 | 426.3 | 120.12 |
| 10485760 | 87.8 | 24.05 | 149.2 | 36.7 | 509.2 | 150.1 | 605.3 | 299.7 |

Table 6 summarizes the variations in computation times or the time spreads of the CPU and GPU implementations for various noise levels used for analyzing the effect with 4 different data sets. The time spreads for the GPU implementations are smaller. This is because of the fact that the GPU time to execute the same operations is significantly lesser than the CPU. In general, the observations made indicate that with data sets consisting of the large number of points, the time spread or variation is bigger. Further, increase in noise also results in bigger time spreads.

The trends in Figure 10 through Figure 18 indicate that as the computational complexity or in other words the arithmetic intensity of operations executed on the graphics hardware increases, the gain in speed increases linearly. Moreover, the nature of the problems being addressed here is such that the number of computations required increases with the increase in the number of points in the data set. The parallel architecture of the graphics hardware and its ability to perform better with increase in computational intensity accounts for the trends observed from the linear regression analysis explained above. Comparison of the CPU and GPU implementations between the torus and circle fit explains this phenomenon. The GPU is about 3 times faster than the CPU in the case of the circle fit where the arithmetic instructions executed on the hardware include population of two columns of the Jacobian and the right hand side vector. In comparison, the GPU implementation for the torus fit which involves computation of six columns of the Jacobian matrix and the right hand side vector is about 14 times faster than the CPU implementation.

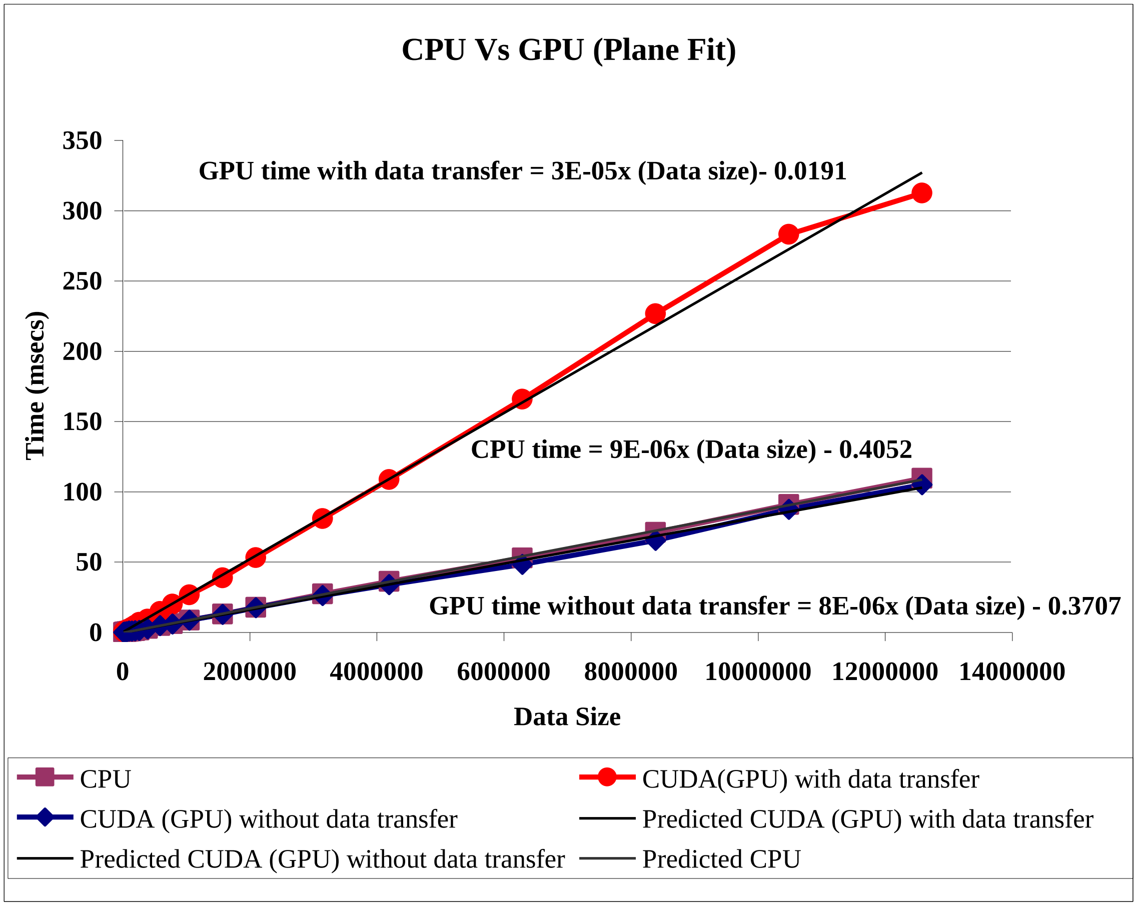

Figure 22: CPU Vs GPU (Plane Fit)

Figure 22 illustrates the computation times for the CPU and GPU implementations of the plane fit based on Table 5. The GPU implementation is benchmarked with and without taking data transfer time into consideration. This is done to demonstrate that the data transfer between the host and the device plays a significant role in the performance. As seen from the plot and also from the relation obtained from the regression analysis the CPU and GPU operations take the same time for almost all the data sets when data transfer time is not considered. The non-linearity discussed in the earlier implementations is present in the case of the plane as well. The trend lines for all the three cases considered have a negative y-intercept. However, the slopes of these lines give a fair estimate of the actual phenomenon observed. The GPU implementation of the translation operation without data transfer is about 1.125 times faster than the CPU implementation which in turn is about 3 times faster than the GPU implementation with data transfer. This is due to the fact that all the threads executing translation operation try to access the same memory location which causes serialization. Unlike all the other implementations, the entire data are copied to the device and back which takes a significant amount of the total computation time.

# CHAPTER VI

# CONCLUSIONS AND RECOMMENDATIONS

## **Conclusions**

The least squares fitting algorithms for analytic geometries namely circle, sphere, plane, cylinder, cone and torus are implemented on a CPU using C++. More specifically the Gauss Newton algorithm as applicable to all non linear geometries has been implemented. Computationally expensive operations in this algorithm are identified and these are implemented on a GPU using CUDA kernel programming. All the implementations are validated for accuracy and are benchmarked with data sets of varying sizes.

An efficient memory model has been adopted for all the GPU implementations and this is shown to achieve a significant reduction in computational time. This is demonstrated for implementations of all primitives except the plane and relevant observations are made. From the observations, it is evident that with increase in arithmetic intensity of operations being executed on the GPU, the performance gain obtained increases significantly. The computations common to all GPU implementations except the plane are population of the Jacobian matrix and right hand side vector which involve executing the same arithmetic instructions on all the data in consideration. Thus, increasing the data size increases the number of arithmetic instructions linearly. Furthermore, the considering of the variety of arithmetic instructions executed in the circle, sphere, cylinder, cone and torus fits, the computational complexity increases further. A comparison is drawn between the performances of the GPU in the least computationally expensive implementation (circle) with that in the most computationally expensive implementation (torus). The GPU is about 3 times faster than the CPU in operation for the circle fit but the factor increases to about 14 for the torus fit. This is largely due to the fact that the parallel processing capabilities of the GPU are exploited efficiently in the case of the torus fit with more number of processors executing the large volume of instructions.

Control flow instructions do not affect performance of the GPU significantly when specified in an efficient manner as demonstrated with the two cases of the cone. It is also observed that a minimum number of points are required in a data set for the GPU to outperform the CPU when the complexity of the arithmetic instructions is not high.

In the case of the plane fit, there is no significant performance gain observed when the GPU is benchmarked against the CPU. The GPU is 3 times slower than the CPU with data transfer and about 1.125 times faster without considering the data transfer time. The reasons for these trends in the results are explained.

However, there are certain limitations with the model and in general with the hardware and software. Some of these are identified as follows

- Instructions cannot be issued and managed directly on the hardware. A CPU (host) platform is required which adds an overhead of data transfer.
- The model adopted for this particular problem limits the mathematical operations that can be executed on the card. Only operations such as transformation and population of Jacobian and the right hand side vector can be performed efficiently.

- The number of data points in a data set should always be in the order of $2^n$ . This is because the number of threads in a block is always a multiple of 16.
- Numerical operations such as matrix multiplication or the QR decomposition as applicable to the problems in this research be performed efficiently. This is due to the fact that the data are processed on blocks which induce a limitation in the control of memory accesses.
- Although the Cholesky decomposition has been previously implemented on graphics hardware[30], in this specific problem the limitations of the hardware and software model do not permit the efficient implementation of this operation. Moreover, the sizes of the positive symmetric definite matrices obtained for these problems are trivial and hence the implementation of this matrix operation is not efficient.
- Debugging in run time is difficult and it is not possible to store or view the intermediate results until the entire kernel program has terminated operation.

## Recommendations

The work carried out in this research and the results obtained demonstrate that Graphical Processing Units (GPUs) are a useful tool in solving least squares fitting of analytic primitives in a significantly less amount of time. The validation of the implementations with actual measurement data can produce significant gain in processing time in coordinate metrology problems.

However, the next generation GPUs can play a very important role in problems such as the one addressed in this work. Feasibility to read and write data directly onto the hardware instead of using a CPU platform, better performance indices can be attained. Furthermore, increased flexibility in programming would allow more number of operations to be handed off to the GPU along with ones already implemented essentially allowing the entire computation to be executed in parallel.

More robust fitting algorithms such as the Levenberg-Marquardt algorithm can be implemented on the GPU. It is possible to check the feasibility of deploying two or more analytic primitive fitting algorithms simultaneously for parts which are a combination of many primitives.

## Contributions

The major contributions in this research include

- Implementations of the non-linear minimization algorithms as applicable to the least squares fitting of geometric primitives (circle, sphere, cylinder, cone, torus and plane) on the CPU in a fairly efficient manner.
- Identification of an efficient memory model to implement specific computations involved in these algorithms on the GPU and its integration with the CPU.
- Benchmarking the CPU as well as the GPU implementations to demonstrate that the parallel processing capabilities of the hardware are exploited in an efficient manner to obtain significant gain in computation times.

# REFERENCES

1. Hopp, T.H., *Computational Metrology.* ASME Manufacturing Review, December 1993. **6**(4): p. 295-304.
2. Lin, S.S., et al., *A Comparative Analysis of CMM Form Fitting Algorithms.* Manufacturing Review, 1995. **8**(1): p. 47-58.
3. Choi, W., *Computational Analysis Of Three Dimensional Measurement Data*, in *Dept. of Mechanical Engineering*. 1996, Carnegie Mellon University: Pittsburgh.
4. Gass, S.I., Witzgall, C., Harary, H. H., *Fitting circles and spheres to coordinate measuring machine data.* International Journal of Flexible Manufacturing Systems, 1998. **10**(1): p. 5-25.
5. Shakarji, C.M., *Reference algorithms for Chebyshev and one-sided data fitting for coordinate metrology*. Cirp Annals-Manufacturing Technology, 2004. **53**(1): p. 439-442.
6. Hopp, T.H., Levenson, M.S., *Performance Measures for Geometric Fitting in the NIST Algorithm Testing and Evaluation Program for Coordinate Measurement Systems.* Journal of Research of the NIST, 1995. **100**(5): p. 563-574.
7. Zwick, D.S., *Applications of orthogonal distance regression in metrology.* Proceedings of the second international workshop on Recent advances in total least squares techniques and errors-in-variables modeling table of contents, 1997: p. 265-272.
8. Pratt, V., *Direct Least Squares Fitting of Algebraic Surfaces.* ACM Computer graphics, 1987. **21**(4).
9. Taubin, G., *Estimation of Planar Curves, Surfaces, and Nonplanar Space-Curves Defined by Implicit Equations with Applications to Edge and Range Image Segmentation.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1991. **13**(11): p. 1115-1138.
10. Keren, D. and C. Gotsman, *Fitting curves and surfaces with constrained implicit polynomials.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1999. **21**(1): p. 31-41.
11. Blane, M.M., Lei, Z. B., Civi, H., Cooper, D. B., *The 3L algorithm for fitting implicit polynomial curves and surfaces to data.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 2000. **22**(3): p. 298-313.
12. Plass, M. and M. Stone. *Curve-fitting with piecewise parametric cubics*. in *International Conference on Computer Graphics and Interactive Techniques*. 1983
13. Sourlier, D. and A. Bucher, *Surface-Independent, Theoretically Exact Bestfit for Arbitrary Sculptured, Complex, or Standard Geometries.* Precision Engineering-Journal of the American Society for Precision Engineering, 1995. **17**(2): p. 101-113.
14. Forbes, A.B., *Least-squares best-fit geometric elements. In Algorithms for approximation II(edited by J. C. Mason and M. G. Cox). London: Chapman and Hall.* 1990: p. 311-319.

15. Ames, A.L., *Cloud to CAD*, in *Sandia Report* SAND, Editor. 2001, Sandia National Laboratories: Albuquerque, NM.

16. Ahn, S.J., Rauh, W., Warnecke, H.J., *Least-squares orthogonal distances fitting of circle, sphere, ellipse, hyperbola, and parabola.* Pattern Recognition, 2001. **34**: p. 2283-2303.

17. Gander, W., Golub, G.H., Strebel, R., *Least-square fitting of circles and ellipses.* BIT Numerical Mathematics, 1994. **43**: p. 558-578.

18. Lukacs, G., D. Marshall, and R. Martin, *Geometric Least-Squares Fitting of Spheres, Cylinders, Cones and Tori*, in *GML*. 1997, Computer and Automation Institute, Hungarian Academy of Sciences: Budapest, Hungary.

19. Lukacs, G., D. Marshall, and R. Martin, *Robust segmentation of primitives from range data in the presence of geometric degeneracy.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 2001. **23**(3): p. 304-314.

20. Shakarji, C.M., *Least-squares fitting algorithms of the NIST algorithm testing system.* Journal of Research of the National Institute of Standards and Technology, 1998. **103**(6): p. 633-641.

21. Atieg, A., Watson, G.A., *Incomplete Orhogonal Regression.* BIT Numerical Mathematics, 2004. **44**: p. 619-629.

22. Atieg, A., Watson, G.A., *A class of methods for fitting a curve or surface to data by minimizing the sum of squares of orthogonal distances.* Journal of Computational and Applied Mathematics, 2003. **158**: p. 277-296.

23. Claudet, A.A., *Analysis of Three Dimensional Measurement Data and CAD Models*, in *Dept. of Mechanical Engineering*. 2001, Georgia Institute of Technology.

24. Chen, A.H. and T.R. Kurfess, *Bounding box techniques to initialize optimization of primitive geometry fitting.* Journal of Manufacturing Systems, 2004. **23**(1): p. 15-21.

25. Olano, M. and A. Lastra. *A Shading Language on Graphics Hardware: The Pixel Flow Shading System*. in *Proceedings of SIGGRAPH*. 1998.

26. Thompson, C.J., S. Hahn, and M. Oskin. *Using Modern Graphics Architectures for General Purpose Stream Computing: A Framework and Analysis*. in *Annual IEEE/ACM International Symposium on Micro Architecture*. 2002.

27. Kruger, J. and R. Westermann, *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*, in *Computer Graphics and Visualization Group*. 2003, Technical University: Munich.

28. Buck, I., et al., *Brook for GPUs: Stream computing on graphics hardware.* Acm Transactions on Graphics, 2004. **23**(3): p. 777-786.

29. Galoppo, N., Govindaraju, N. K., Henson, M., Manocha, D. *LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware*. in *Supercomputing, Proceedings of the ACM/IEEE* 2005.

30. Jung, J.H., *Cholesky Decomposition and Linear Programming on a GPU,* in *Dept. of Computer Science*. 2006, University of Maryland.

31. III, K.E.H., et al. *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*. in *International Conference on Computer Graphics and Interactive Techniques*. 1999. Chapell Hill.

32. Rumpf, M. and R. Strzodka. *Using Graphics Cards for Quantized FEM Computations*. in *Proceedings of Visualization, Imaging and Image Processing Conference*. 2001.

33. Wu, E., Y. Liu, and X. Liu, *An Improved Study of Real Time Fluid Simulation on GPU*, Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences: Beijing, China.

34. Hillesland, K.E., S. Molinov, and R. Grzeszczuk, *Nonlinear Optimization Framework for Image-Based Modeling on Programmable Graphics Hardware.* Proceedings of ACM SIGGRAPH 2003, 2003. **22**(3): p. 925-934.

35. Khardekar, R., G. Burton, and S. McMains, *Finding feasible mold parting directions using graphics hardware.* Computer-Aided Design, 2006. **38**(4): p. 327-341.

36. Gray, P.J., F. Ismail, and S. Bedi, *Graphics-assisted Rolling Ball Method for 5-axis surface machining.* Computer-Aided Design, 2004. **36**(7): p. 653-663.

37. Roth, D., Ismail, F., Bedi, S., *Mechanistic modeling of the milling process using an adaptive depth buffer.* Computer-Aided Design, 2003. **35**: p. 1287-1303.

38. Roth, D., Ismail, F., Bedi, S., *Mechanistic modeling of the milling process using complex tool geometry.* The International Journal of Advanced Manufacturing Technology, 2005. **25**: p. 140-144.

39. Pabst, H., Springer, J.P., Schollmeyer, A., Lenhardt, R., Lessig, C.; , *Ray casting of trimmed NURBS surfaces on the GPU.* IEEE symposium on Interactive Ray Tracing, 2006.

40. *NVIDIA CUDA Compute Unified Device Architecture*, in *Programming Guide Version 0.8*. 2007, NVIDIA.