Hole Patching In Unstructured Mesh And Parallelization Using Graphics Processing Units

Amitesh Kumar
University of Alabama at Birmingham

HOLE PATCHING IN UNSTRUCTURED MESH AND PARALLELIZATION USING GRAPHICS PROCESSING UNITS

by

AMITESH KUMAR

A DISSERTATION

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

BIRMINGHAM, ALABAMA

2011

HOLE PATCHING IN UNSTRUCTURED MESH AND PARALLELIZATION USING GRAPHICS PROCESSING UNITS

AMITESH KUMAR

DOCTOR OF PHILOSOPHY IN INTERDISCIPLINARY ENGINEERING

ABSTRACT

Engineering analysis of a three-dimensional geometric model using mesh-based computational technologies requires the model to be topologically watertight. However, achieving watertight geometry is considered to be a challenging task in the field of computational engineering due to the potential presence of geometric deficiencies, such as gaps and holes on the surfaces. This dissertation aims to repair the defective geometric model with the presence of holes irrespective of their complexities. Presented in this dissertation are novel research and implementation of a hybrid surface and volume-based technique for geometry repair. It utilizes a NURBS-based surface-patching algorithm for topologically simple holes and incorporates a volumetric hole-patching algorithm for complex holes. The volume-based hole-patching algorithm solves the diffusion equation using an explicit forward difference scheme in time and a centered difference scheme in space. A robust and efficient algorithm has been developed to both identify and extract a localized hole region. An automated mesh generation process has been implemented to construct individual “column grids” for each isolated hole region. The diffusion equation is solved using finite-difference techniques to generate a scalar solution field from which isosurfaces are extracted with an isovalue that represents the repaired surfaces for the local regions. Finally, a Poisson surface reconstruction is used to create a reconstructed watertight surface.

The graphics processing unit (GPU) has emerged as the most powerful chip in a computer in the last decade but has only in the past few years received extensive attention from the research community for its use in high performance computing. This research explores a GPU-based implementation of a diffusion equation solution to better harness its computation potential and to facilitate the computational needs of geometry repair. Comparisons of the speedup gains for diffusion solutions using GPGPU with that of conventional single and multi-threaded implementations are presented, and their performance characteristics are discussed in this dissertation.

Dedicated to my parents

ACKNOWLEDGEMENTS

It would be almost impossible to acknowledge everyone who directly or indirectly contributed to the completion of my dissertation work. I would like to apologize to those whose names I might miss in this acknowledgement, but your contributions are never the less really appreciated.

First and foremost, I would like to express my deepest gratitude to my adviser and chair of my dissertation committee, Dr. Alan Shih, for his constant support and for always being very patient with me. I would like to express my sincere thanks to Dr. Roy Koomullil, Dr. Yasushi Ito, Dr. Purushotham Bangalore and Dr. David Thompson for serving on my committee and for giving me invaluable suggestions during the course of my research. I am also thankful to the current and former programmer staff members of the Enabling Technology Laboratory (ETLab) and the UAB Department of Mechanical Engineering, especially Mark Dillavou, Doug Ross, Corey Shum and Fredric Dorothy, for their assistance and great suggestions.

I am also very thankful to my dearest friends Abby Becker, Ankit Srivastava, Ashutosh Ranjan, Eli Johnson, James Lambert, Jessica Ford, Jon Becker, Keith Gugliotto, Michelle Michael and Ravi Bharadwaj for always being there for me and for keeping my morale high during the course of my studies and my stay in Birmingham.

NVIDIA supported my research through generous hardware donations, and I wish to thank Mr. Stan Posey of NVIDIA for facilitating that critical donation. I would also like to express my gratitude for the financial support of NASA's Constellation University Institutes Project (CUIP) program for the early part of my graduate study that established the foundation of my knowledge in numerical geometry. My participation in the National Science Foundation Innovative Technology Experiences for Students and Teachers (ITEST) program provided me with valuable K-12 technology education and outreach experience, and the financial support for the last 2 years of my Ph.D. study, for which I am equally grateful.

I would like to express my appreciation towards the Stanford 3D Scanning Repository and AIM@SHAPE Shape Repository for providing public domain validation models for me to use in this study, such as “Stanford Bunny,” “Chinese Lion,” and “Laurent Hand.” I would also like to express my sincere gratitude s towards Laurent Saboret et al. for making the “Laurent Hand” and “Chinese Lion” models publicly available for research.

Finally, this acknowledgement would not be complete without remembering the contribution of my family members, especially my parents, Mr. U.S. Agrawal and Mrs. Champa Agrawal, and my uncle, Dr. Vinay Kumar. Without their support and encouragements, I could never have been where I am today.

Amitesh Kumar

9/05/2011

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

3D	Three Dimensional
AMD®	Advanced Micro Devices
API	Application Process Interface
CAD	Computer Aided Design
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CSM	Computational Solid Mechanics
CUDA	Compute Unified Device Architecture
GGTK	Geometry and Grid Toolkit
GPGPU	General Purpose Computation using GPUs
GPU	Graphics Processing Unit
NURBS	Non-Uniform Rational B-Spline
VTK	Visualization Toolkit

CHAPTER 1

INTRODUCTION

The preprocessing steps in simulations using mesh-based computational technologies, such as Computational Fluid Dynamics (CFD) and Computational Structural Mechanics (CSM), involve geometry preparation and mesh generation. However, it has been a challenging task to turn geometry into a high-quality mesh when the geometry is complex. Moreover, the potential presence of geometrical deficiencies such as gaps or holes, protrusions or intersections, and overlaps further complicates the process, as a topologically watertight geometric model is required in the meshing process. Geometry repair can be a laborious and complicated process, and the configuration of the geometry can make the process of satisfactory geometry repair difficult. Therefore, how to obtain a watertight geometry ready for the mesh generation process is an important issue in computational engineering.

An interactive geometry creation using mesh generation tools directly is a preferred approach, as this can reduce the conversion errors that introduce some of the geometry defects between the Computer-Aided Design (CAD) systems and the mesh generation tools. However, most of the mesh generation tools do not have sophisticated solid modeling capabilities when compared to those of CAD systems. As a result, geometries for most sophisticated real-world applications are first produced on CAD systems in parametric form and then imported into mesh generation tools tailored for the needs of downstream applications. Geometries can also be sourced in discrete forms. For example, scanned data generated from range-finding devices such as laser scanners can produce a fairly accurate geometric model defined by a point clouds, which can be turned into discrete elements that represent the surface model of the object. Geometry can also be reconstructed from segmented contours of image slices from image-based, patient-specific biomedical data. Regardless of the source of the geometry data, or the form in which they are represented (parametrically or discretely), the geometries obtained can have many defects due to the problems at the source, data conversion errors or ambiguities in the process. A surface mesh of a geometric model is considered to be non-watertight (or non-manifold) in the following two cases:

- It has edges that are shared by only one polygon, i.e., when the edges lie on the boundary. The occurrences of such a set of connected boundary edges create a hole on the surface.
- It has edges which are shared by more than two polygons. This kind of non-manifold mostly occurs in CAD applications due to the improper stitching of a surface patched to generate a desired geometric model.

How to repair defective geometric surface models automatically and robustly while maintaining the high fidelity of the geometric information remains a critical area of research in the field of computational geometry. This research addresses surface defects of the first kind, i.e., holes in discrete geometry, by providing a method for mesh repair by the numerical solution of the diffusion equation for volumetric domain. This dissertation presents an automatic and robust hole patching (or hole filling) algorithm for geometrically and topologically complex holes. The outcome of this research impacts the field of computational engineering, which is widely utilized in the aerospace, automotive, mechanical and biomedical engineering communities.

This work presents a volume-based mesh repair algorithm in chapter 4. The volume-based algorithm could be used to repair surface meshes with holes. The chapter describes the process of identifying and isolating surface defects such as holes and isles from the original surface, which are later used to create individual volume-mesh *solution columns*. These *solution columns* are created by embedding regions of interests in Cartesian grids and switching on the voxels, which intersect with triangles on the input surface mesh. It is assumed that one side of the embedded geometry in the voxelized *solution column* is heated and the other side is cold. A finite volume-based diffusion equation solver is iterated on the voxelized Cartesian mesh until the solution achieves convergence. The solution so obtained represents a scalar field in space which would have closed the gaps existing in the initial input mesh in the voxelized Cartesian grid. The voxelized scalar fields are contoured using a Marching Cubes algorithm to obtain surfaces in each of the *solution columns*. Once the resultant surfaces are obtained from various *solution columns* surrounding the hole regions of the geometry, they are used to create consistently oriented point sets using contouring technique on the converged solution. The point sets, in turn, are used to generate a reconstructed watertight geometry using Poisson Surface Reconstruction [36], [38]. The results of this mesh repair process are presented with a number of examples and plots in chapter 4.

Furthermore, a hybrid mesh repair technique is described in chapter 5 that uses both surface-based and volume-based mesh repair techniques. The surface-based mesh repair technique, used in the hybrid method, builds on the work described in Kumar et al. [12], Kumar and Shih [13], and Kumar et al. [14] with improvements. Surface-based mesh repair technique is followed by the use of volume-based mesh repair technique and finally Poisson reconstruction to obtain a watertight reconstructed model. The reconstructed model so obtained may be substantially different from the original input model. It was found that the hybrid method provides superior quality results when compared with either only surface-based or volume-based mesh repair methods for complicated geometries. This was demonstrated with examples.

The diffusion equation solution process presented in this research is tied to the resolution of the input mesh. If the input mesh is of higher resolution, then this would cause the *solution column* for the same hole on the surface of the mesh to be a larger size when compared with that of an input mesh of lower resolution. This would imply that the diffusion solver would spend a far longer time in trying to find a convergent solution for the diffusion equation. As a result, an effort to improve the computational efficiency of the diffusion solver through parallelization was also made in this research. The diffusion equation solution process has been parallelized both on the CPU and GPU architectures. The parallelization of the diffusion solver on CPU has been done by multi-threading using the OpenMP library [68]. The parallelization on GPU was done for the NVIDIA GPU using CUDA [51] to study the speedup of the solution process when compared with that on the CPU architecture. The results from this study are presented in chapter 6 with a number of tables and plots.

Most of the work being presented in this research was performed on a desktop with a 64-bit quad-core AMD® Athlon™ II 620 processor and 4GB of RAM. The GPU is a single NVIDIA® Quadro™ FX 5800 graphics card, with 240 stream processor cores and 4GB of onboard GDDR5 graphics memory, and was generously donated by NVIDIA for this research. This dissertation only presents the GPU-related studies and conclusions for NVIDIA GPU and any discussion about GPGPU-related development work in here should be considered synonymous with the work done on the NVIDIA GPU.

CHAPTER 2

GEOMETRY REPAIR

Computer-Aided Engineering (CAE) plays an important role in design, analyses, and performance predictions, given the rapid advances made in computer hardware performance and software algorithms. High Performance Computing (HPC) software tools are nowadays widely deployed in government laboratories, academic institutes and industries. Mesh-based CAE disciplines, such as Computational Fluid Dynamics (CFD) and Computational Structure Mechanics (CSM), facilitate the design and analysis processes as well as significantly reduce costs associated with design and development. These technologies are heavily relied upon to produce performance data of complex configurations involving complicated multi-physics processes. However, before any CFD or CSM algorithms can be applied to analyze a given design, a high-quality mesh must be generated. The mesh generation process in turn relies on the availability of a watertight geometry. Unfortunately, such watertight geometry may not always be available due to deficiencies, such as gaps and holes. This is sometimes true even for engineering geometries designed with a sophisticated Computer-Aided Design (CAD) system. Discrete geometry acquired through a reverse engineering process or geometry reconstruction is even more likely to have such deficiencies. “Repairing” such defective geometry is often a tedious and labor-intensive process. It involves removing defects or artifacts from a geometric model to produce an output model that is suitable for further processing by downstream applications with certain input quality requirements.

The most common type of mesh defects or artifacts encountered are holes or isles, singular vertex, handle, gaps and small overlaps, large overlaps, inconsistent orientation, complex edges and intersections. Some of these artifacts, such as complex edges, have a precise meaning, while other artifacts are described intuitively rather than definitively, such as the distinction between small scale and large. Figure 1 shows the major artifacts that occur in meshing as presented in Botsch et al. [1].

A significant amount of research has been done and published in an attempt to address this issue in a more automated and intelligent manner. These approaches broadly fall in two main categories: surface-based repair methods and volume-based repair methods.

Surface-based Repair Methods

This class of repair methods operates directly on the input data and tries to explicitly identify and resolve artifacts on the surface. For example, gaps could be removed by snapping boundary elements (vertices and edges) onto each other or by stitching triangle strips between the gap. Holes can be closed by a triangulation that minimizes a certain error term. Intersections could be located and resolved by explicitly splitting edges and triangles.

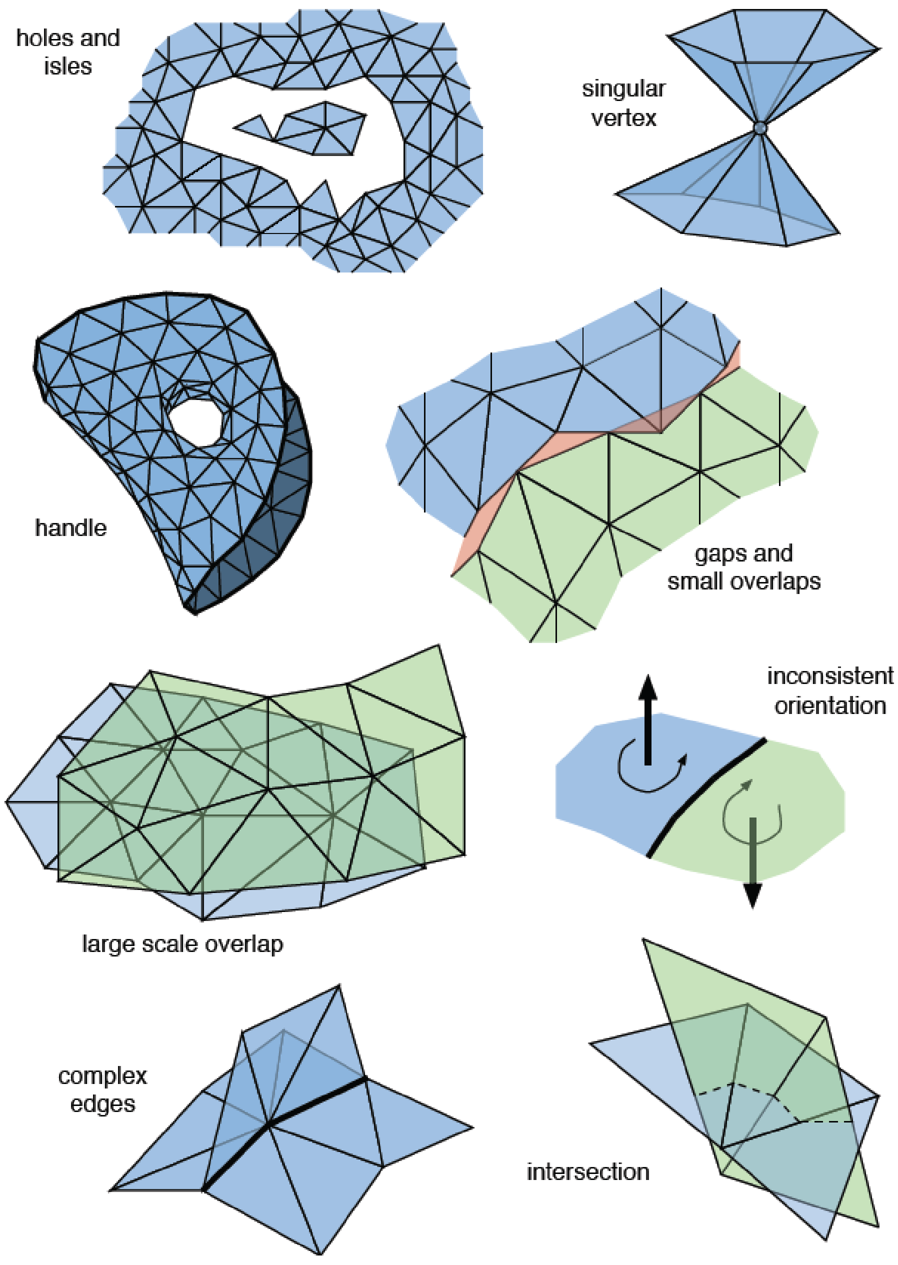


Figure 1: Artifact Chart

Note: From “Geometric Modeling Based on Triangle Meshes” by Botsch M, Pauly M, Rössl C, Bischoff S and Kobbelt L (2006) ACM SIGGRAPH 2006 Courses, article 1: pp 34. Adapted with the permission of the author.

Surface-oriented repair algorithms only minimally perturb the input model and are able to preserve the model structure in areas that are not near artifacts. In particular, structure encoded in the connectivity of the input (e.g., curvature lines) or material properties associated with triangles or vertices are usually well preserved. Furthermore, these algorithms introduce only a limited number of additional triangles.

Surface-oriented repair algorithms usually require that the input model already satisfy certain quality requirements such as error tolerances to be able to guarantee a valid output. Since these requirements cannot be guaranteed or even be checked automatically, as a result these algorithms are rarely fully automatic and require manual post-processing. Furthermore, due to numerical inaccuracies, certain types of artifacts, e.g., intersections or large overlaps, cannot be resolved robustly. Other artifacts, e.g., gaps between two closely connected components of the input model that are geometrically close to each other, are difficult to identify, as described by Botsch et al. [1].

A number of algorithms have been suggested for filling holes in a triangular mesh using a surface-based repair approach. Turk et al.'s mesh-zippering algorithm [2] is one of the first algorithms which tried to fuse range images using a surface-based approach and in this process eliminated a number of the overlaps and mesh defects. This algorithm, however, thus specializes for range-scan data.

Barequet and Sharir [3] use a dynamic programming method to find the minimum area of triangulation for a three-dimensional (3D) polygon in order to fill holes. Barequet and Kumar [4] describe an interactive system that closes small cracks by stitching corresponding edges and fills big holes by triangulating the hole boundary, similar in approach to Barequet and Sharir [3]. Their methods, in general, are optimized for the defects generated by the CAD programs while joining the surfaces to create models, which sometime leaves narrow cracks. Although these methods may work well in patching narrow holes, they may fail where simple stitching may not provide an elegant solution. These algorithms also do not guarantee the quality of the output.

Gueziec et al. [5] propose a method to remove complex edges and singular vertices from non-manifold input models. Their work claimed to generate an output which is guaranteed to be a manifold triangle mesh, possibly with boundaries. Their algorithm operates solely on the connectivity of the input model and as such does not suffer from numerical robustness issues. In a pre-processing phase, all complex edges and singular vertices are identified. The input is then cut along these complex edges into manifold patches. Finally, pairs of matching edges, i.e., edges which have the same endpoint, are identified and, if possible, merged.

Gueziec et al. [5] introduce two different strategies for stitching edges left unstitched by pinching and snapping. The pinching strategy only stitches along edges that belong to the same connected component. The small, erroneous connected components are separated from the main part of the model and could be detected and removed in a post-processing step. In contrast to pinching, the snapping strategy reduces the number of connected components of the model. Their basic idea is to locate candidate pairs of boundary edges and to stitch them if, after stitching, the model does not contain new complex edges or singular vertices. The scope of their algorithm is limited to the removal of complex edges and singular vertices.

Guskov and Wood [6] propose an algorithm that detects and resolves all handles up to a given size in a manifold triangle mesh. Handles are removed by cutting the input mesh along a non-separating closed path and sealing the two resulting holes by triangle patches. Their algorithm reliably detects small handles up to a user-prescribed size and removes them. However, the algorithm is slow, does not detect long, thin handles and cannot guarantee that no self-intersections are created when a handle is removed.

Borodin et al. [7] propose a progressive gap-closing algorithm which works by vertex edge contraction accompanied with insertion of vertices on the boundary edges and progressively contracting the edge. This is implemented by identification of corresponding *vertex-vertex* pairs and *vertex-edge* pairs. This method, although simple in implementation, is only suitable for narrow gaps in 3D space where such contraction does not end up dramatically altering the surface smoothness and triangle size gradation.

Leipa [8] describes a method for filling holes by a weight-based hole triangulation, mesh refinement based on the Delaunay criterion and mesh fairing based on energy minimization as used in Kobbelt et al. [25]. It builds on the works of Klincsek [26] and Barequet and Sharir [3]. The algorithm could only be used for filling holes in an oriented, connected mesh. The algorithm reliably closes holes in models with smooth patches, with the density of the vertices in the filled area matching that of the surrounding surface. The complexity of building the initial triangulation is $O(n^3)$, which is sufficient for most holes that occur in practice. However, the algorithm does not check for or avoid self-intersections and does not detect or incorporate isles into the hole-filling process.

Jun [9] describes an algorithm based on a stitching planar projection of a complex hole and projecting back the stitched patch. This method presents significant complexity of implementation if the holes are twisted and if their intermediate projections onto a surface are self-intersecting. The resultant patch produced in this manner also may not look very elegant or smooth.

Branch et al. [10] suggest a method for filling holes in triangular meshes using a local radial basis function. The method works quite well with small and narrow holes but is not as successful when the holes are large compared to the size of the nearby features and are rounder in shape. The usability of most of these algorithms, with the notable exception of Leipa [8], is constrained by their assumptions related to the shape, size or source of the holes.

Kumar et al. [12] describe a surface-based hole-patching algorithm which produces smooth patches using an innovative concentric ring-based approach around the holes or gaps in triangulated meshes. They use NURBS curves and NURBS surfaces to create smooth patches, and the reliability of their method is dependent on the reliability of the initial triangulation algorithm used in 3D space. The density of the vertices in the generated patch matches that of the average density of the surrounding vertices in the neighborhood of the holes. This algorithm only repairs the meshes with topologically simple holes and does not detect or incorporate isles into the filling. Their method is further improved in Kumar and Shih [13] and Kumar et al. [14] by introducing a smoothing technique at the interface of the patches and boundaries of the holes.

Volume-based Repair Methods

The key to all volume-based methods lies in converting a surface model into an intermediate volumetric representation from which the output model is then extracted.

Examples of volumetric representation that have been used in model repair include regular Cartesian Grids, adaptive octrees, kd-trees, BSP-trees and Delaunay triangulations. A flag at each voxel of the volumetric representation is generated specifying whether the particular voxel lies inside, outside, or on the surface of the geometry. The interface between inside and outside cells defines the topology and geometry of the reconstructed model. Due to their very nature, volumetric representations do not allow for artifacts such as intersections, holes, gaps or overlaps or inconsistent normal orientation. Volumetric algorithms are typically fully automatic and produce watertight models and, depending on the type of volume, they can often be implemented very robustly as described by Botsch et al. [1].

Volume-based approaches to mesh repair also pose some potential problems. The conversion to and from a volume leads to a resampling of the model. It often introduces aliasing artifacts, loss of model features and destroys any structure that might have been present in the connectivity of the input model. The number of triangles in the output of a volumetric algorithm is usually much higher than that of the input model and thus has to be decimated in a post-processing step. Also, the quality of the output triangles often is degraded and has to be improved afterwards. Finally, volumetric representations are quite memory intensive so it is hard to run them at high resolutions. If the fidelity of the data is of utmost importance, then one might want to consider a surface-based approach, which respects the triangulation on the original mesh and hence preserves the original data in this process.

Voxelization of a surface mesh requires a method to accurately define box-triangle intersections for every triangle on the surface mesh with voxels in the volume. Akenine-

Möller [15] presents a fast 3D Triangle-Box overlap testing method based on the separating axis-theorem. The theorem states that two convex polyhedra, A and B, are disjoint if they can be separated along either an axis parallel to a normal of a face of either A or B, or along an axis formed from the cross product of an edge from A with an edge from B. The paper focused on an axis-aligned box (AABB) to find the intersection of a triangle and the box. The source code for AABB intersection is in public domain and is freely available [16].

Curless and Levoy [18] and Davis et al. [19] propose one of the most well-known methods to repair a mesh using a volumetric approach. In their methods, the inside-outside flags are generated with the help of a distance map of each point on the geometry using line-of-sight information, which is usually obtained from range-finding devices. This crucial piece of information may not be available for a purely computational geometric model. The uncertain voxels are assigned flags based on volumetric diffusion. Once all the voxels are assigned flags, the volume-based methods simply extract the contour to find a closed surface. Curless and Levoy's method [18] was optimized for data obtained from range-finding devices. These devices generally create very high resolution computational models which may contain holes with complex topologies due to occlusion as well as surface reflections and refractions.

Nooruddin and Turk [20] propose one of the first volumetric techniques to repair arbitrary models containing gaps, overlaps and intersections. In this method, the model is first converted into a Cartesian voxel grid using *parity-count* and *ray-stabbing* methods. Most of the volumetric mesh-repair methods propose the use of an inside-outside flag to classify whether a voxel lies inside or outside of the intermediate volume representation of the original surface model. In this method, a set of projection directions $\{d_i\}$ is produced by subdividing an octahedron or icosahedron. Then the model is projected along these directions onto an orthogonal planar grid. For each grid point x, the algorithm records the first and last intersection point of the ray $x + d_i$ and the input model. A voxel is classified by such a ray to be inside if it lies between these two extreme depth samples; otherwise, it is classified as outside. The final classification of each voxel is derived from the majority vote of all the rays passing through that voxel. A Marching Cubes algorithm [40] is then used to extract the surface between the inside and outside voxels.

Nooruddin and Turk [20] further take advantage of the common morphological operators, *dilation* and *erosion*, used in 3D digital image-processing techniques as low pass filters to fill small gaps and tubes on the intermediate volume representation. Rafael et al. [17] provide a good description of different morphological techniques currently being used in image processing.

Ju [21] presents a method for generating the signs of voxels for repairing a polygonal mesh using an adaptive Octree approach. Ju mentions that the method, although simple in conception and design, may not be able to produce satisfactory results for those cases that have complex holes with multiple boundaries or highly curved shapes. His algorithm produces guaranteed manifold output by virtue of using a volumetric method; however, the algorithm seemed to have a problem handling thin structures. Due to the volumetric representation, the whole input model is resampled, and the output may also become arbitrarily large for fine resolutions in this process.

Bischoff et al. [22] propose an improved volumetric technique to repair arbitrary triangle soups using a user-provided error tolerance value ε and a maximum diameter value ρ up to which gaps should be closed. Their algorithm first creates an adaptive octree representation of the input model in which each cell stores the triangles intersecting with it. From these triangles, a feature-sensitive sample point can be computed for each cell. Then a sequence of morphological operations [17] is applied to the octree to determine the topology of the model. The connectivity and geometry of the reconstruction are derived from the octree structure and samples, respectively. Finally, a *Dual Contouring* algorithm then reconstructs the interface between the outside and the inside cells by connecting sample points. These sample points minimize the squared distances to their supporting triangle planes. As a result, it is claimed that features like edges and corners are well preserved. If no such planes are available, the corresponding sample point is smoothed in a post-processing step.

Podolak et al. [23] propose an algorithm for 3D hole filling based on a decomposition of space into atomic volumes, which are each determined to be either completely *inside* or *outside* of the model. It is done by computing a minimum-cost cut of a graph representation of the atomic volume structure that is guaranteed to produce non-intersecting patches.

Murali and Funkhouser [27] present a unique method for converting triangle soups to manifold surfaces. In their method, the polygon soup is first converted in a *Binary Space Partition* (BSP) tree while the supporting planes of the input polygon serve as the splitting plane for partitioning space into a set of polyhedral regions. This helps to determine which regions are solid, based on region adjacency relationships. They claim that, unlike other approaches, their solid-based approach is effective even when the input polygons intersect, overlap, are wrongly-oriented, have T-junctions, or are unconnected.

Although their method doesn't need any user parameter to automatically produce watertight models, the output may also contain complex edges and singular vertices which may require further post-processing.

Surface Reconstruction

Surface reconstruction is a widely studied topic in the area of computer geometry and computer graphics. There are several approaches to surface reconstruction based on global and local approaches. Many of these reconstruction methods are based on combinatorial structures, such as Delaunay triangulation, alpha shapes, and Voronoi diagrams, which try to interpolate the surface on all or most of the input points. Other schemes try to directly represent that surface in implicit forms. Global methods which use surface fitting in implicit forms often need to solve extremely large, dense and ill-conditioned matrices. Local fitting methods, on the other hand, try to consider only a subset of input data at a time and try to create local radial basis functions (RBF) to define tangent planes. These methods face a number of difficulties due to non-uniformity in the sampling of data, presence of noise as well as missing input data.

Amenta and Bern [28] present an algorithm for reconstructing an interpolating surface from sample points in 3D space using Voronoi filtering and the β-skeleton. Their reconstruction is based on the definition of a planar graph on sample points called the "crust" and proves to be highly sensitive to the local geometry, point sampling, and any noise which might be present in the input point set.

Amenta et al. [29] present a surface reconstruction algorithm called “The Power Crust,” which constructs a piecewise-linear approximation to object surface and their medial axis transform for the input points. Their method uses a set of Voronoi diagrams which divides space into polyhedral cells. The union of the outer faces of these polyhedral cells provides the reconstructed surface.

Carr et al. [30] present a method using polyharmonic radial basis functions to reconstruct a manifold surface from point-cloud data and sometimes repair undersampled surfaces with voids. Their method consists of three different steps which includes : construction of a signed-distance function, fitting an RBF to the resulting distance functions and, finally, iso-surfacing the fitted RBF.

Bruno [31] attempts to fill a hole and blend surface-based on global parameterization for complete geometry approximation and then energy minimization for surface blending based on the assumption that global parameterization of the complete model is available or possible.

Dey and Goswami [32] present a method called “Tight Cocone,” which guarantees a watertight output surface without introducing any extra points based on the peeling of tetrahedras. The tetrahedras are created based on Delaunay reconstruction from 3D simplexes in 3D space using input points.

Shen et al. [34] propose a volumetric repair algorithm that operates on arbitrary triangle soups. Their algorithm makes use of a scattered-data interpolation method known as *moving least-squares* (MLS) with a number of constraints that forces the function to give a value of the surface region for each polygon. The degree of approximation, in their method, is controlled by adjusting the least-squares weighting function. The tightness of the surface, or in other words, the requirement of input vertices falling inside the implicit surfaces, depends on an iterative procedure for adjusting the constraint values over each polygon.

Kazdan et al. [36] present a surface reconstruction method using oriented points based on the solution of spatial Poisson equations. Their reconstruction method calculates a 3D indicator function from the input points. The indicator function is then used to extract an isosurface. Poisson reconstruction is claimed to be more tolerant to noisy, non-uniform input data compared to other reconstruction algorithms, and the results in support of their conclusion were included in their paper. A number of open source implementations of their methods are available. This research uses an implementation of Poisson surface reconstruction developed by Doria and Gelas [38] for the VTK library.

Isosurface and Contouring Methods

An isosurface is a three-dimensional (3D) surface that represents points of a constant value of a scalar field variable, such as pressure, temperature, velocity, density and intensity within a volume of space. In other words, it is a level set of a continuous function whose domain is 3D-space. Isosurfaces are used in computer graphics, data visualization, and medical imaging, among a number of other areas. A watertight isosurface is sometimes also used to generate a 3D grid by growing 3D tetrahedras from the 2D triangles present on the isosurface. There are a number of contouring methods available, such as Marching Cubes, Dual Marching Cubes, Extended Marching Cubes and Adaptive Marching Cubes, among others. Marching Cubes is a highly successful contouring algorithms which can create triangulated models with constant density isosurfaces from volumetric data. However, the Marching Cubes algorithm is not without its shortcomings, such as a lack of feature sensitivity, aliasing artifacts, high triangle density, etc. Regardless, the Marching Cubes algorithm [40] is used in this research for contouring purposes due to the availability of its implementations. As a result, it becomes necessary to discuss the theory behind this contouring method.

The Marching Cubes algorithm was first presented by Lorenson et al. [40]. The algorithm uses a two-step approach to isosurface construction problems. The first step involves locating the surface corresponding to the user-defined value. The second step deals with calculating a normal to the surface at each vertex of the cube. Marching Cubes uses a divide-and-conquer approach to locate the surface in a logical cube created from eight pixels, four each from two adjacent slices.

The algorithm determines how the surface intersects this cube and then marches to the next cube. To find the surface intersection in a cube, a value of one is assigned to a cube's vertex if the data value at that vertex equals or is greater than the value of the surface that is being constructed. These vertices are either on the inside or on the surface. Cube vertices with values below the surface receive a zero and are outside the surface. The surface intersects only those cube edges which have one vertex outside (zero) and another inside the surface (one). With this assumption, the topology of the surface within the cube is determined, with the locations of the intersections found later. Since there are eight vertices in a cube and two states inside or outside, it provides $2^8 = 256$ ways a surface can intersect a cube. However, looking at all the combinations, one realizes that by using a permutation of complementary and rotation symmetry of the cube, the number of choices reduces from 256 patterns to a total of 14 topologically unique cases and one case where none of the edges intersect the surface. A unique index is created for each of the cases based on the state of the vertices. The index is n 8-bit number with the lowest significant bit representing the state of vertex 1 while the highest significant bit represents the state of vertex 8. The index so formed is used to lookup from a list of pre-calculated tables. The surface intersection along the edge is found using linear interpolation. In the final step, a unit normal over each cube vertex is found by a central difference method, and the normal is interpolated to find the normal value at each of the triangle's vertices. This provides a triangulated contoured surface for a 3D voxelized data.

CHAPTER 3

GPU COMPUTING

Modern graphics processing units (GPUs) have emerged as the most powerful chip in high performance workstations with increasing parallelism rather than increasing clock rate as the primary engine of processor performance growth. The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaces its CPU counterpart. Unlike multi-core CPU architectures, which currently ship with up to eight cores, GPU architectures are multi-core, with hundreds of cores capable of running thousands of threads in parallel. This degree of hardware parallelism reflects the fact that GPU architectures evolved to fit the needs of real-time computer graphics, a problem domain with tremendous inherent parallelism. Figure 2 shows the evolution of NVIDIA GPU vis-à-vis that of INTEL CPUs in terms of theoretical peak performance in gigaflops per second or GFLOP/s. The fast-paced development of the GPU, in the recent past, has been spearheaded by the rapid development of a massive computer gaming industry with its onerous demands of more realistic, high-resolution rendering at a very high frame rate.

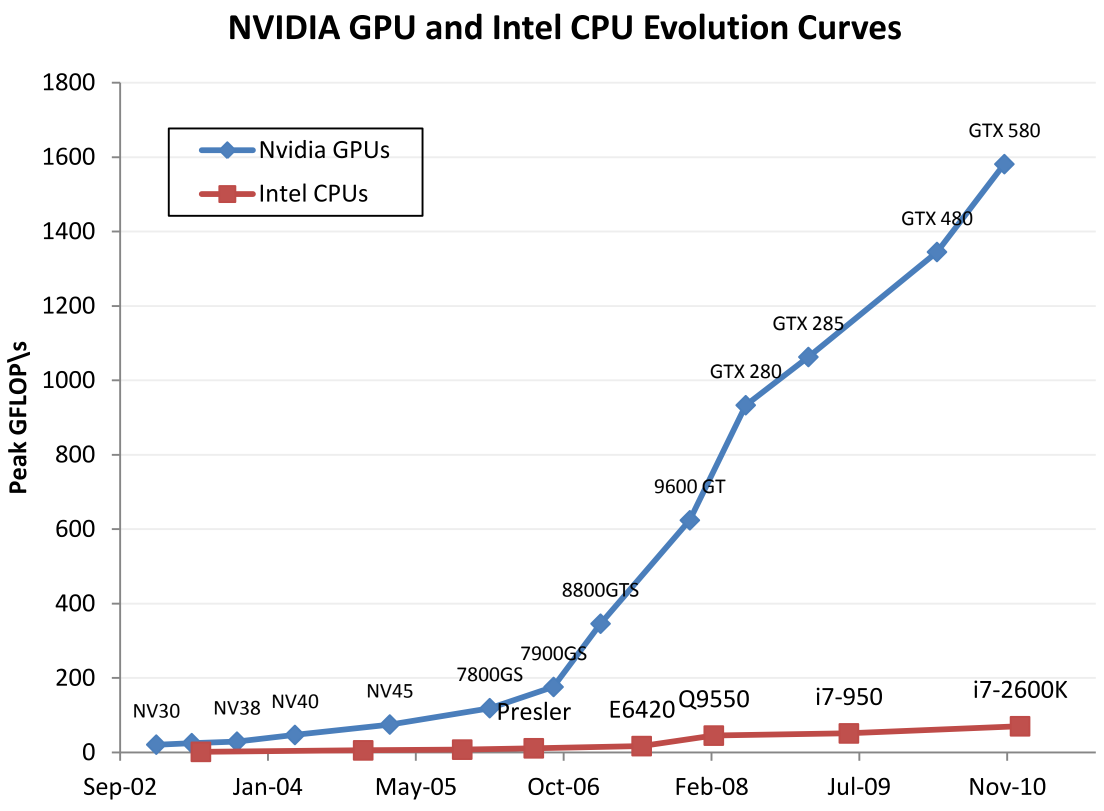


Figure 2: NVIDIA GPU and Intel CPU Evolution Curves

The GPU’s rapid increase in both programmability and capability has spawned a research community that has successfully mapped a broad range of computationally demanding, complex problems to the GPU. Around 1999-2000, GPU peak performance was catching up with that of the CPU, and the research community started taking notice of it. As a result, some basic experimentation started to test the suitability of GPU for general purpose scientific computing. It was found that the excellent floating point performance in GPUs could lead to a huge performance boost for a range of scientific applications [51]. This effort in general purpose computing on the GPU, also known as GPGPU, has positioned the GPU as a compelling alternative to traditional microprocessors in high-performance computer systems of the future. A typical GPU has evolved for applications having following characteristics, as described in Owens et al. [43]:

- Large computational requirement*:* Real-time rendering on billions of pixels per second, with each pixel requiring a hundred or more operations at a high frame rate.
- Substantial parallelism*:* Programmable compute units in GPUs are inherently parallel in nature and are, in general, very suitable for operations on vertices and fragments.
- Emphasis on greater throughput than latency*:* Emphasis in any modern processor is placed on maximum performance through higher throughput rather than latency due to the orders of magnitude difference between the human visual system and an operation within a modern processor.

The Graphics Pipeline

The input to the GPU is a list of geometric primitives, typically triangles, in a 3D world-coordinate system. Through many steps, those primitives are shaded and mapped onto the screen, where they are assembled to create a final picture. These steps are as follows:

- Modeling transformations*:* The graphics primitives are transformed from the object-coordinate system to the world-coordinate system, which would later be mapped onto the display screen. This is done by the coordinate transformation of individual vertices with a single transformation matrix, which is the product of many modeling transformation matrices representing various geometry operations.

- Per vertex lighting and vertex operations: The input primitives are formed from individual vertices. Each vertex is transformed into screen space and shaded by computing its interaction with the lights in the scene.
- Assembly: The vertices are assembled into triangles, which are the fundamental hardware-supported primitives in today's GPUs.
- Rasterization: This process determines the correlation between screen pixel locations and the triangles covering them. Each triangle generates a primitive called a *fragment* at each screen-space pixel location that it covers. Because of the possibility of multiple overlaps of *fragments* at any pixel location, each pixel's color value may be computed from several fragments.
- Fragment Operations: Each fragment is shaded with the fetched color and texture information from global memory to determine its final color. This stage is typically the most computationally demanding stage of the graphics pipeline and is designed to run in parallel.
- Composition: This stage is the final assembly of screen pixels into a screen image with one color per pixel location and is done by keeping the closest pixel to the camera for each pixel.

A typical scene has tens to hundreds of thousands of vertices, triangles and fragments and each of the operations in the graphics pipeline can be computed independently. Hence, these operations are well suited for parallel hardware and are performed efficiently due to the massively parallel architecture of the modern GPU.

GPU Architecture

A few years ago, GPU was a fixed-function processor, built around a graphics pipeline, especially designed to do only a few things related with primitive graphics operations but do those efficiently. In legacy GPU architecture, a GPGPU computing task needed to be masked as a graphics program with the real computation masked as a series of vertex operations and pixel shading calculations. No matter how efficient legacy GPUs may have been, their fixed function pipeline architecture made it difficult to use the GPU chip for any purpose other than rendering images. The real path toward General Purpose GPU computing began, not with GPUs, but with onboard programmable 3D graphics accelerators. Multi-chip 3D rendering engines were developed by several companies starting in the 1980s. But by the mid 1990s, it became possible to integrate all the essential elements onto a single chip with the rapid advances in chip design and breakthroughs in chip fabrication technologies. From 1994 to 2001, these chips progressed from the simplest pixel-drawing functions to a full implementation of the 3D pipeline including geometry transforms, vertex operations like lighting, rasterization, fragment operations such as coloring and texturing and finally composition into frame buffer with depth testing and display.

In 2001, NVIDIA’s GeForce 3 introduced programmable pixel shading to the consumer market for the first time. The programmability of this chip was very limited, but later GeForce products became more flexible and faster, adding separate programmable engines for vertex and geometry shading. This evolution culminated in the NVIDIA’s GeForce 7800 series of graphics cards, which was based on their G70 chip and was first released commercially in early 2006. It had higher bandwidth and higher peak performance when compared with its predecessors and cutting- edge CPUs of the time as presented in Figure 2. It had several improvements over its predecessors with its 3D pipeline and additional stages of configurable and fixed function logic, which could be programmed in the context of graphics applications.

GPGPU programming evolved as a way to perform non-graphics processing on these graphics-optimized architectures, typically by running carefully crafted shader code against data presented as vertex or texture information and retrieving the results from a later stage in the pipeline. GPGPU programming showed great promise. However, managing multiple programmable engines in a single 3D pipeline present on the GPU led to bottlenecks, as too much effort went into balancing the throughput of each stage [44].

In November 2006, NVIDIA introduced the G80 chip-based GeForce 8800 series of GPU products and later the GT200 extended the performance of G80-based architecture. This design featured a unified shader architecture with 128 processing elements distributed among eight shader cores. Each shader core could be assigned to any shader task, eliminating the need for stage-by-stage balancing and greatly improving overall performance. The main features of the G80 architecture in support of GPU computing were as as follows:

- It was the first GPU to support the C programming language.
- It was the first GPU to replace separate vertex and pixel pipelines with a single unified processor that executed vertex, geometry, pixel and computing programs.
- It introduced a single-instruction, multiple-thread execution model where multiple independent threads execute concurrently using a single instruction.

- It introduced shared memory and barrier synchronization making thread synchronization issues trivial for GPU-based general purpose computing.

In April 2010, NVIDIA introduced a GPU based on the Fermi Architecture. It was a significant improvement over previous generations of G80 and GT200-based GPU designs. The main improvement in GPUs based on Fermi architecture when compared with those based on the G80 and GT200 design are support for C++ programming language constructs on the GPU, 32 CUDA cores per streaming processor, true double precision, ECC support on its high end Fermi based graphics card, true cache hierarchy, more shared memory, faster context switching between different threads, and faster atomic operations.

The fixed-function pipeline on the older GeForce 7800 series of GPUs lacked the generality to express efficiently more complicated shading and lighting operations that are essential for complex effects. The GeForce 8800 series of GPUs replaced the fixed-function per-vertex and per-fragment operations with capability to run user-specified programs on each vertex and fragment. Over the last couple of years, these vertex programs and fragment programs have become increasingly more capable, with larger limits on their size and resource consumption, more fully featured instruction sets, and more flexible control-flow operations. Current GPUs support the Unified Shader Model 4.0 standard on both vertex and fragment shaders as mentioned in Owens et al [44].

In the legacy GPUs at the hardware level, the operations available at the vertex and fragment stages could be configured but could not be programmed. In the fixed-function pipeline, the programmer could control the position and color of the vertex and the lights, but not the lighting model that determined their interaction. As the shader model has evolved and become more powerful and GPU applications of all types have increased vertex and fragment program complexity, GPU architectures have increasingly focused on the programmable parts of the graphics pipeline. Thus, over the past few years, the GPU has evolved from a fixed-function, special-purpose processor into a full-fledged parallel programmable processor with some additional fixed-function special-purpose functionality.

As both the vertex and fragment programs became more fully featured with demand for more realistic rendering and as the instruction sets converged, GPU architects decided in favor of a unified shader architecture against the previous strict task-parallel pipeline based architecture. The benefit for the newer GPU unified shader architecture is better load-balancing at the cost of more complex hardware. Due to the unified shader architecture, GPGPU computing has become even more rewarding with all the programmable power in a single hardware unit. GPGPU programmers can now target that programmable unit directly and optimize their tasks on them, rather than the previous approach of dividing work across multiple hardware units which created performance bottlenecks.

GPU Programming Model

The programmable units of the GPU follow a single program multiple-data (SPMD) programming model. For efficiency, the GPU processes many elements (vertices or fragments) in parallel using the same program. Each element is independent from the other elements, and in the base programming model, elements cannot communicate with each other. All GPU programs must be structured in this way: many parallel elements each processed in parallel by a single program. Each element can operate on 32-bit integer, floating point or even double precision data in the latest generation GPUs with a reasonably complete general-purpose instruction set. Elements can read data from a shared global memory (a *gather* operation) and, with the newest GPUs, also write back to arbitrary locations in shared global memory (*scatter*).

One of the benefits of the GPU is its large fraction of resources devoted to computation when compared with that of CPU. Today's CPU spends a significant portion of its silicon real estate to allow a different execution path for each element. Instead, today's GPUs support arbitrary control flow per thread but impose a penalty for incoherent branching. In a GPU, elements are grouped together into blocks and blocks are processed in parallel. If elements branch in different directions within a block, the hardware computes both sides of the branch for all elements in the block [44]. In writing GPU programs branches are permitted but are somewhat cost prohibitive.

Programming a GPU for General-Purpose Programs

Programming GPGPU applications has been historically difficult, since despite their general-purpose tasks having nothing to do with graphics, the applications still had to be programmed using APIs hidden in the graphics. The general purpose programs had to be structured in terms of the graphics pipeline as vertex and texture operations, with the programmable units only accessible as an intermediate step in that pipeline. Today, GPU computing applications are structured in such a way that the programmer directly defines the computation domain of interest as a structured grid of threads, and an SPMD general-purpose program computes the value of each thread.

The value for each thread is computed by a combination of mathematical operations and both *gather* (read) accesses from and *scatter* (write) accesses to global memory. Direct access of the programmable units to the global memory eliminates much of the complexity faced by previous GPGPU programmers in co-opting the graphics interface for general-purpose programming. As a result, GPGPU programs today are more often expressed in a familiar programming language, such as NVIDIA's C-like syntax in their Compute Unified Device Architecture (CUDA) programming environment or in OpenCL so that ports are simpler and easier to write, execute and debug. This results in a programming model that allows its users to not only take full advantage of the GPU's powerful hardware, but permits an increasingly high-level programming model that enables productive authoring of complex applications.

NVIDIA and Compute Unified Device Architecture (CUDA)

NVIDIA has been designing and positioning its GPUs as versatile devices suitable for much more than electronic games and 3D graphics. NVIDIA's Tesla brand GPUs are specifically marketed for high-performance computing. Its Quadro brand is marketed for professional graphics workstations while the GeForce brand is intended for the low- end traditional consumer graphics market. Current generation GPUs of any of the segments from NVIDIA can, however, be used for general purpose computing using its CUDA software platform. NVIDIA's CUDA is a software platform that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, Direct Compute and other languages. It was formally introduced in 2006. CUDA requires programmers to write special code for parallel processing. It does not require them to explicitly manage threads in the conventional sense, which greatly simplifies the programming model. It should be noted that although CUDA was first released in the 2006 for programmers and the developer community, all the capabilities which CUDA offers today are only available for a select few new generation GPU devices offered by NVIDIA. As an example, CUDA when used for G80- and GT200-based GPUs, such as the one used in this study, do not provide C++ language support. That feature is only available for the late model GPUs based on Fermi Architecture. The GPU used in this research is a dedicated NVIDIA Quadro FX5800 with 240 onboard stream processors and 4GB GDDR3 memory, as shown in Figure 3. The graphics card is based on GT200 architecture and connects to the CPU bus through a PCI Express 2.0 x16 slot interface.

CUDA includes C/C++ software development tools, function libraries, and a hardware abstraction mechanism that hides the GPU hardware from developer. Although CUDA requires programmers to write special code for parallel processing, it doesn’t require them to explicitly manage threads in the conventional sense, which greatly simplifies the programming model. CUDA development tools work alongside a conventional C/C++ compiler, so programmers can mix GPU code with general-purpose code for the host CPU. Figure 4 shows a CUDA software architecture stack. It has common C/C++ source code with different compiler forks for CPUs and GPUs with function libraries that simplify programming and a hardware abstraction mechanism that hides the details of GPU architecture from programmers.

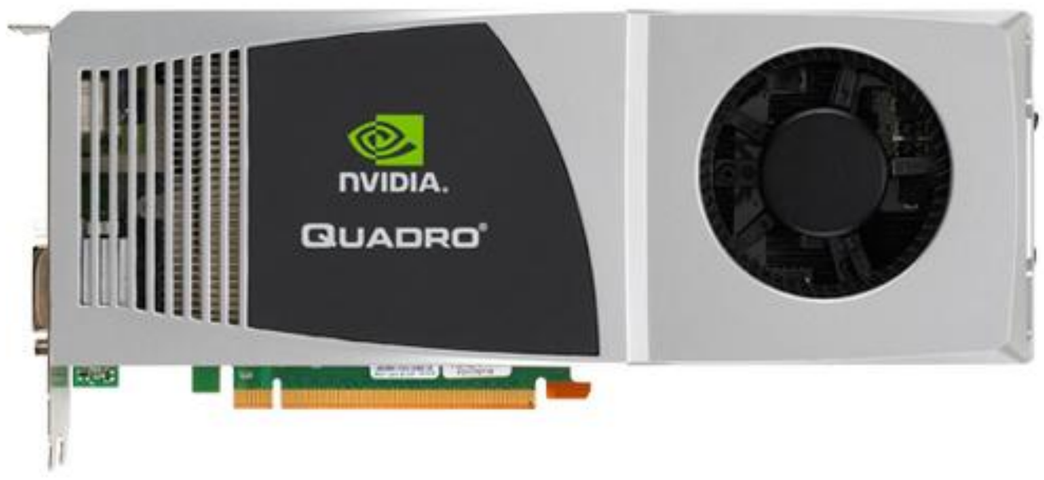
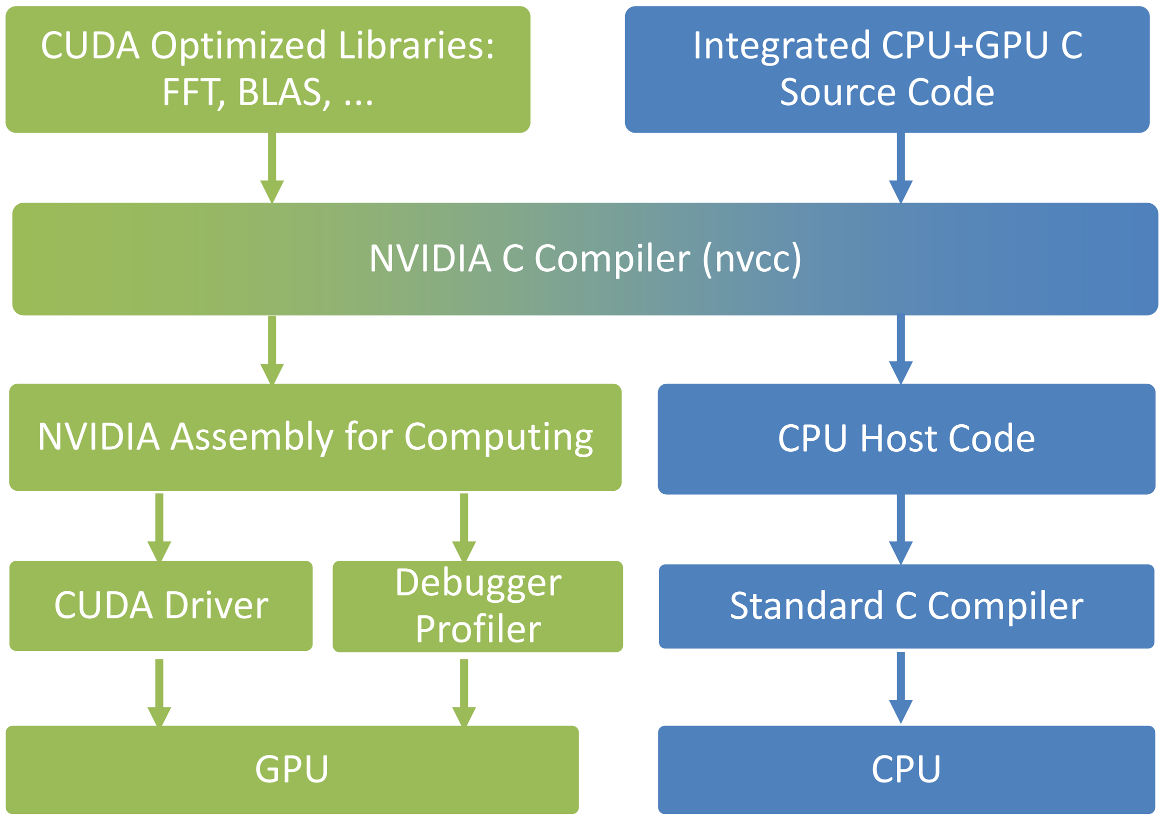


Figure 3: NVIDIA Quadro FX 5800 Graphics Card



CUDA Optimized Libraries: FFT, BLAS, ...
Integrated CPU+GPU C Source Code
NVIDIA C Compiler (nvcc)
NVIDIA Assembly for Computing
CPU Host Code
CUDA Driver
Debugger Profiler
Standard C Compiler
GPU
CPU

Figure 4: NVIDIA's CUDA Software runtime architecture stack

A hardware abstraction model like the one offered by CUDA has two major benefits. First, it simplifies the high-level programming model, insulating programmers from the complex details of the GPU hardware while at the same time letting them take advantage of the benefits offered by GPU hardware architecture for GPGPU programming. The second benefit is that hardware abstraction allows NVIDIA to change the GPU architecture as often as it wants. NVIDIA is free to design processors with any number of cores, any number of threads, any number of registers, and any instruction set. As a result of this abstraction, theoretically a C/C++ source code written today for an NVIDIA GPU using CUDA can run without modification on future NVIDIA GPUs with additional thread processors, or on a future NVIDIA GPUs with a completely different architecture.

In a single-threaded model, the CPU fetches a single instruction stream that operates serially on the data. Single-instruction multiple data (SIMD) extensions permit many CPUs to extract some data parallelism from the code, but the practical limit is usually three or four operations per cycle even for the most efficient CPU per core. Although CUDA’s programming approach is highly parallel, it requires the division of the dataset into smaller chunks stored in on-chip memory allowing multiple thread processors to share each chunk for high performance. Storing the data locally reduces the need to access off-chip memory, thereby improving performance. In the CUDA model, off-chip memory accesses usually don’t stall a thread processor. Instead, the stalled thread enters an inactive queue and is replaced by another thread that’s ready to execute. When the stalled thread’s data becomes available, the thread enters another queue that signals that it is ready to go. Groups of threads, also known as warps, take turns, ensuring that each thread gets execution time without delaying other threads. The efficient and effortless switching of threads in warps, where threads do not have to wait on other threads fetching data, is one of the main reasons for the huge performance gain while running programs on a GPU.

A CUDA program is organized into a host program, consisting of one or more sequential threads running on the host CPU, and one or more parallel kernels that are suitable for execution on a parallel processing device like the GPU. A kernel executes a scalar sequential program on a set of parallel threads. The programmer organizes these threads into a grid of thread blocks. The threads of a single thread block are allowed to synchronize with each other via barriers and have access to a high-speed, per block shared on-chip memory for inter-thread communication. Threads from different blocks in the same grid can coordinate only via operations in a shared global memory space visible to all threads. CUDA requires that thread blocks be independent, meaning that a kernel must execute correctly regardless of the order in which blocks are run. This restriction on the dependencies between blocks of a kernel, although cumbersome, provides scalability.

Developers write kernels in CUDA that execute on the GPU and define a single thread of execution's behavior. Thousands of such threads execute a kernel concurrently, and the GPU's thread manager maps them all to physical thread processors. The kernel is invoked on the host side, at which time the host CPU determines how many threads to execute. The host CPU also controls memory management and data transfer. A special NVIDIA compiler called `nvcc` translates kernels and host programs into code that executes on both the CPU and GPU. CUDA architecture treats threads independently of each other but actually executes them on a single instruction, multiple data- (SIMD) type architecture.

An important feature of CUDA is that application programmers do not write explicitly threaded code. A hardware thread manager handles threading automatically. Automatic thread management is vital when multithreading scales to hundreds of thousands of threads. Although these are lightweight threads in the sense that each one operates on a small piece of data, they are fully fledged threads in the conventional sense. Each thread has its own stack, register file, program counter, and local memory, as described by Halfhill [41]. The GPU preserves the state of inactive threads and restores their state when they become active again. By removing the burden of explicitly managing threads, NVIDIA simplifies the programming model and eliminates a whole category of potential bugs. Even though CUDA automates thread management, it does not entirely relieve developers from thinking about threads and thread management. Developers must analyze their problem to determine how best to divide the data into smaller chunks for distribution among the thread processors based on their GPU architecture. For GPU-based programming, they also need to be aware of the optimal numbers of threads and blocks that will keep the GPU fully utilized and provide maximum throughput for their intended application. Factors affecting performance of their GPU code may include the size of the global data set, the maximum amount of local data that blocks of threads can share, the number of thread processors in the GPU, and the sizes of the on-chip local memories.

A geometric model can have a significant number of holes, either as clusters or far away from each other, that require patching individually. The volume-based hole patching algorithm, presented later in this study, uses the diffusion equations as presented in Appendix A, followed by contouring to generate a smooth continuous surface in the areas near discontinuity. The solution domain is Cartesian and the solution process is repetitive in nature performed independently on a large number of voxels. The size of the solution domain for the diffusion equation is tied to the size of elements in the input mesh surrounding holes in the implementation presented as part of this study. As a result, when the input mesh becomes finer, the solution domain becomes bigger. As a result the diffusion equation solver spends even more time in finding a convergent solution to the diffusion equations. The repetitive nature of the solution process and a large number of computations for finding a convergent solution of diffusion equation in a Cartesian domain makes it an ideal problem for GPU based parallelization. The research work presented in this study parallelizes the diffusion solver for both GPUs and multi-core CPUs and would compare and contrast their performance gain with respect to the single-core CPU run time in Chapter 6 using CUDA toolkit and library. The solution process would use a single NVIDIA Quadro FX 5800 GPU on CentOS linux operating system.

CHAPTER 4

VOLUME APPROACH TO GEOMETRIC HOLE PATCHING

Surface-based hole-patching algorithms provide good quality mesh repair. In general, they minimally alter the surrounding geometry. However, surface-based algorithms usually require that the input model already satisfy certain quality requirements such as clean geometry, no intersections or overlaps, etc., to be able to guarantee a valid output. Many of these requirements cannot be met or even be checked automatically. Furthermore, due to numerical inaccuracies, certain types of artifacts, such as intersections or large overlaps, cannot be resolved robustly. Other artifacts, like gaps between two closed connected components of the input model that are geometrically close to each other, cannot even be identified as described in Botsch et al. [1].

Volumetric representations, on the other hand, do not allow for artifacts like intersections, holes, gaps or overlaps or inconsistent normal orientation. Volumetric algorithms are typically fully automatic and can produce watertight geometries. They can often be implemented very robustly. As a result, for some cases, it is necessary to use a volume-based approach to repair models. Figure 5 presents a flow chart of the mesh repair process using a volume-based algorithm, which will be discussed in this chapter in detail.

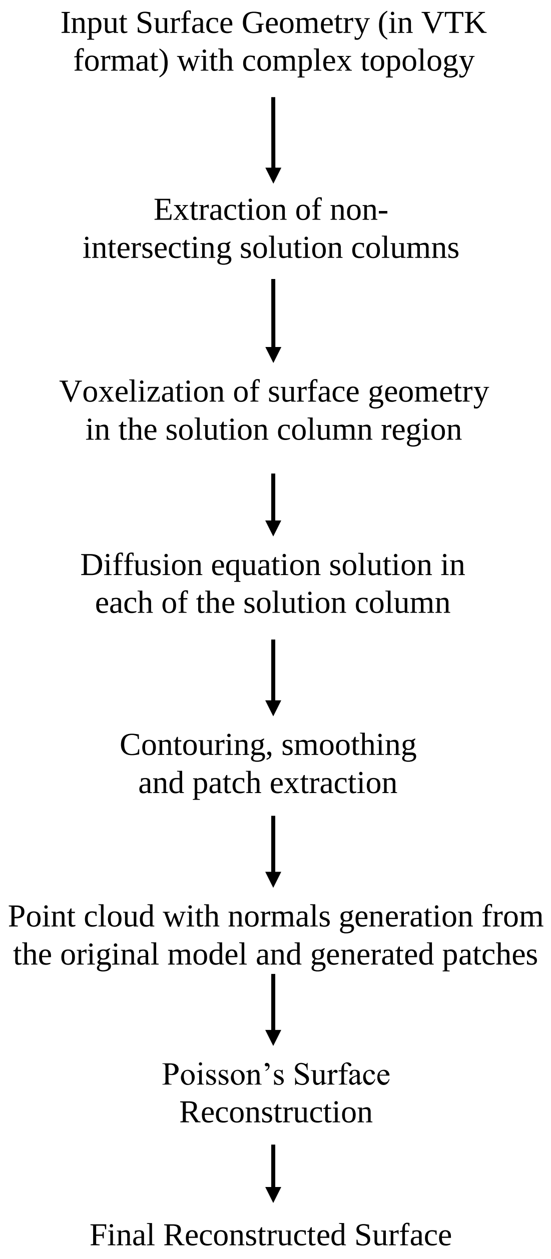

Input Surface Geometry (in VTK format) with complex topology
Extraction of non-intersecting solution columns
Voxelization of surface geometry in the solution column region
Diffusion equation solution in each of the solution column
Contouring, smoothing and patch extraction
Point cloud with normals generation from the original model and generated patches
Poisson's Surface Reconstruction
Final Reconstructed Surface

Figure 5: Flowchart of volume-based and hybrid approach towards mesh repair.

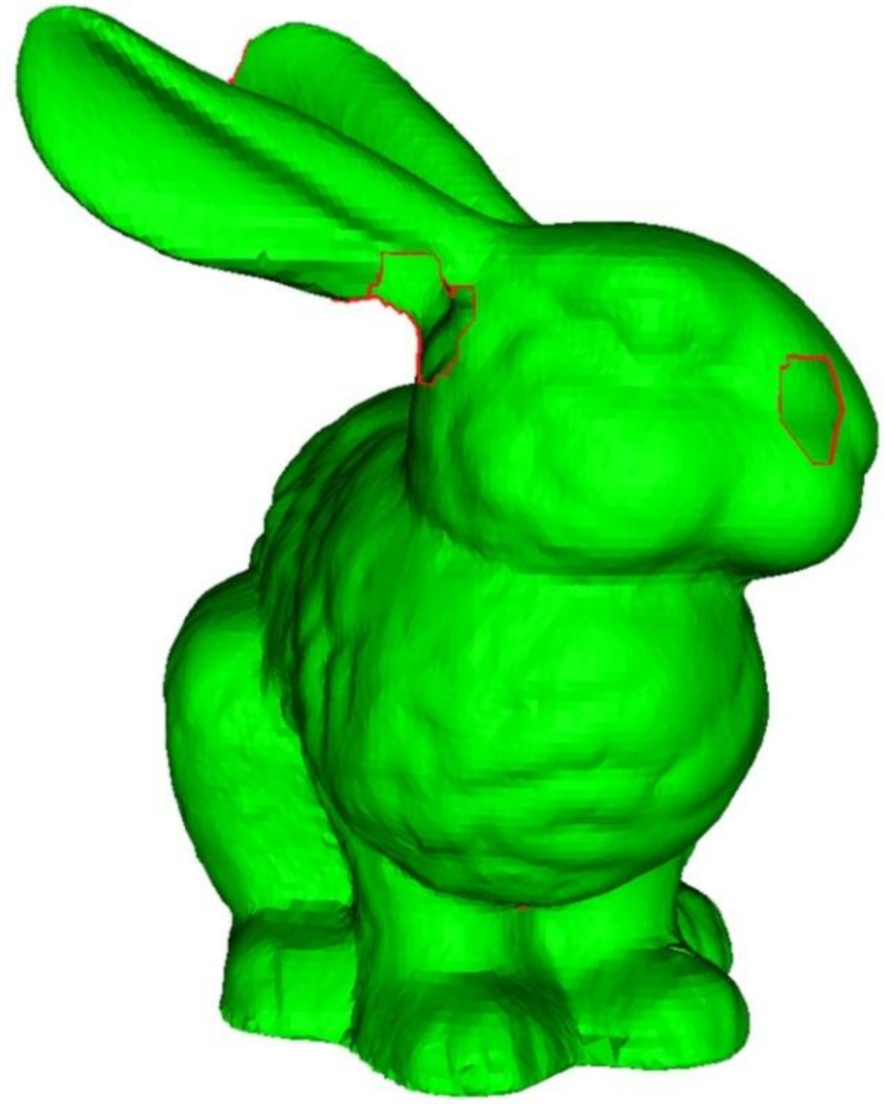


Figure 6: Stanford Bunny model with 10 holes

This volume-based approach being presented is loosely based on previous work done by Davis et al. [19], with some improvements. A fully automatic volume-based repair method is presented in this chapter. It can handle a dirty geometry with holes, isles and small intersections. The Stanford Bunny [57] model is used to illustrate the results of the study in this chapter.

Stanford Bunny

The Stanford Bunny [57] is one of the most commonly used validation models in computer geometry and computer graphics. It is freely available to the community for research purposes. The Stanford Bunny was a product of the Digital Michelangelo Project [67] executed in Stanford's Computer Graphics Laboratory. It was obtained by assembling a number of scanned range images of a clay bunny roughly 7.5 inches tall using a Cyberware 3030 MS scanner. The model so obtained is a collection of 69,451 triangles and 35,947 points. Due to occlusion, the Stanford Bunny has five holes left after assembling the scanned range images. Five additional holes of various sizes were cut in this model for algorithm validation in this research, making a total of 10 holes, as shown in Figure 6.

Extraction of Solution Columns and Voxelization of Discrete Geometry

The first step in volume-based repair method is to convert the surface mesh into a volumetric mesh. In this research, Cartesian grids have been used for their simplicity of implementation. Cartesian grids are generated for the regions of interest of the input geometry to represent the data. However, regions of interest need to be first identified to be able create representative Cartesian grids.

In the previous studies [12], [13] and [14], the input surfaces were considered to have simple topologies, and holes on the input surface were found by finding a set of connected edges which were non-manifold in the sense that the edges belonged to only one polygon. The input surface geometry could, however, be of complex topology and might contain numerous surface fragments along with a largest surface among them. In this study, the largest surface is identified and is separated from the rest of the surface fragments. To identify regions of interest, holes on the largest surface of the mesh are found. The regions of interest are non-intersecting regions which may enclose one or more of the nearby holes and are bigger than the bounding box of the enclosed holes. Each of these regions of interest is termed a "*solution column*" in this dissertation. Each of the *solution columns* is represented as a uniform Cartesian grid. The voxelization is only performed in non-intersecting regions of interest near the surface defects. The fragments and surfaces enclosed within the region of interest are later on embedded in the *solution columns*.

The density of the voxels is a function of the triangle size in terms of average edge lengths and average area of the neighborhood triangles in the region of interest of the input geometry. The density of voxels will not only determine the rate of convergence of the solution but also the distribution of points and the quality of output at the reconstructed surface in the later stages of the mesh repair process. The Cartesian grid can be arranged in the form of a block of 3D tiles, as shown in Figure 7. The user can specify the tiles' size along the maximum length, which determines the number of tiles used to represent the Cartesian grid.

Each tile is of identical size and is composed of a certain number of voxels (*floating precision*) based on the bounding box size. The tiles are padded with an extra layer of ghost cells along the boundary, as shown in Figure 8. The memory allocation is completely dynamic, and the information exchange between the contiguous tiles is hidden from the user using an abstraction layer.

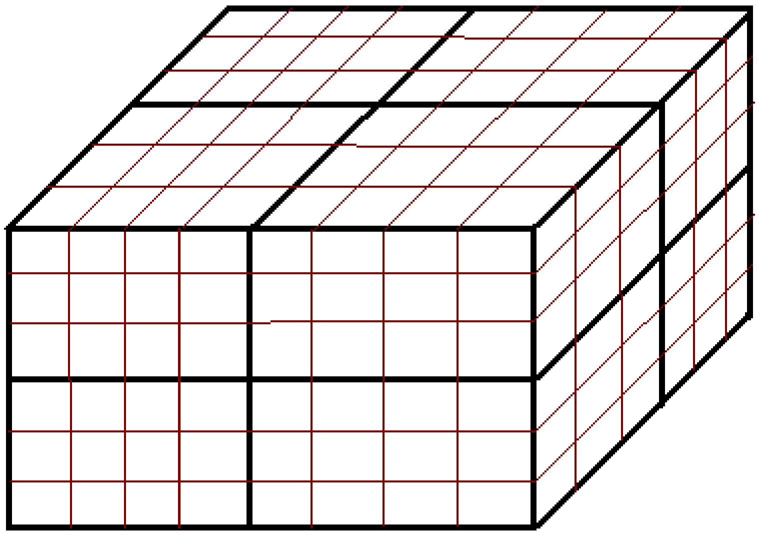


Figure 7: A Cartesian Grid composed of multiple blocks in 3D

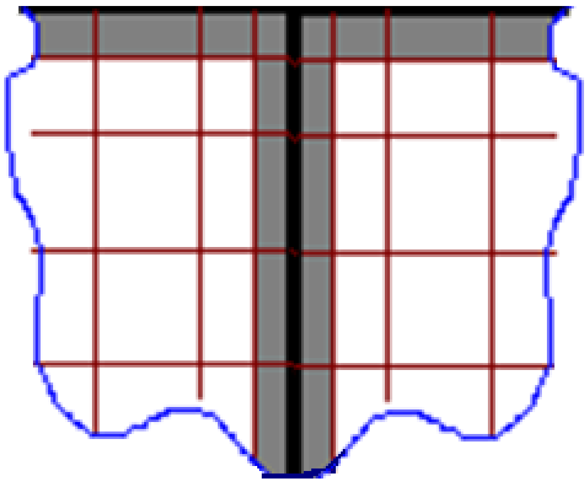


Figure 8: Ghost cells at the interface of two tiles shown in gray color in a 2D Cartesian grid.

Although this representation is more complex than the scenario when the whole data is represented as one single block, this kind of memory allocation for representation of a Cartesian grid has two major benefits:

- Depending on the memory footprint of a tile and the size of the cache on the CPU chip, one may be able to employ cache effects by taking advantage of the cache hierarchy to speed up the code execution. This is possible because the small memory size of each tile may fit inside the cache of the CPU.
- The solver requires an exact temporary copy of the tile in the intermediate step to compute and transfer data. If a small tile size is being used, then only a small intermediate amount of memory would be needed to compute and transfer data for each tile.

As the information exchange between ghost cells is completely hidden from the user, the user would not notice any difference while using the APIs compared to the situation when a whole block is composed of a single tile. A mask is also created for the embedded data using an identical Cartesian grid of lower precision (*char*) to store information from the original model and to store the boundary conditions for the solver. The lower precision of the mask is used to conserve the memory footprint of the solution process. The mask is also used after contour extraction to determine whether a triangle in question is a new triangle in the void region or if it overlaps or intersects with the original triangles of the input region of interest. An input discrete geometry is composed of individual triangles. The intersection of each triangle with the voxels of the Cartesian grid lying within its bounding box is checked using the AABB Triangle-Box intersection algorithm [15]. All voxels that intersect with triangles are masked as "*model voxels*" with a static

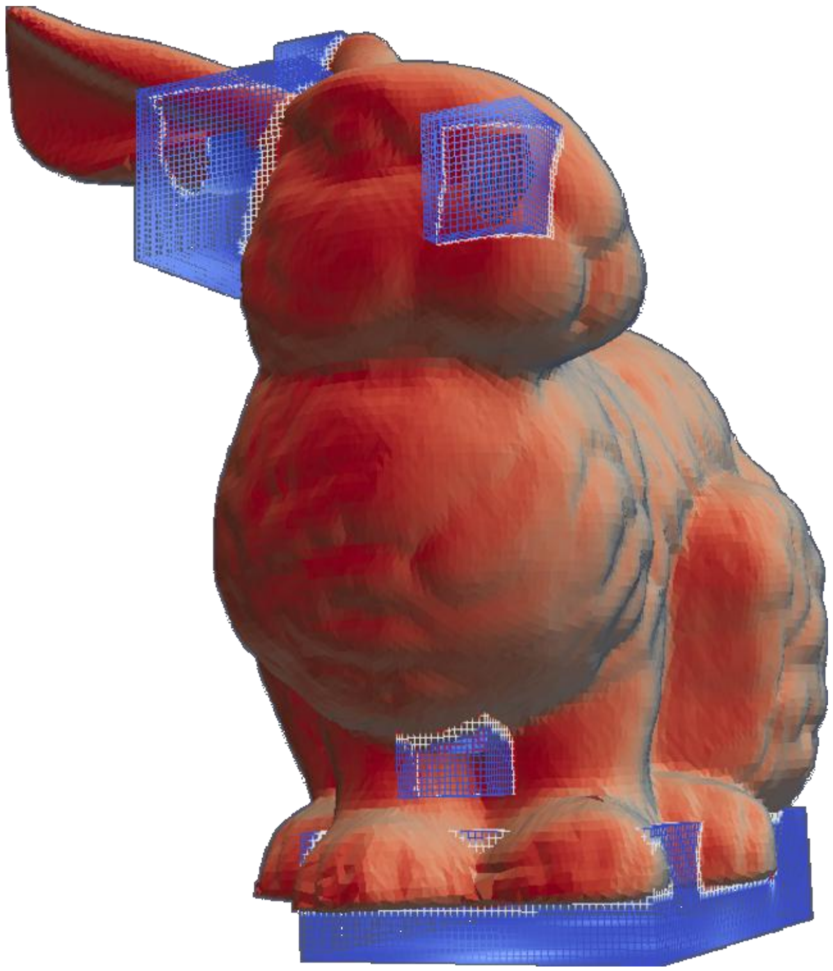


Figure 9: Solution columns for Stanford Bunny

value of ‘0’. All the voxels along the positive normal of the triangle plane which are not “*model voxels*” are masked as “*heated*” and given a static positive value of ‘+1’. All the voxels along the negative normal of the triangle plane which are not “*model voxels*” and are touching “*model voxels*” are masked as “*cold*” and given a negative value of ‘-1’. This process creates a signed Cartesian grid with voxelized representation of the discrete geometry having a neutral value of ‘0’ sandwiched between the hot and the cold sides. The rest of the uncertain voxels are dynamic and can change values during the solution phase. Davis et al. [19] use line-of-sight information obtained from the ranging devices to create a signed data for the solution process.

The algorithm presented in this dissertation is not tailored for input data from ranging devices; hence the availability of line-of-sight information is not assumed in this work, which could be used to sign the voxels of the Cartesian grids. Instead, an assumption is made that the input discrete geometry has consistent normal orientation for the purpose of defining the boundary conditions. The input geometry may not have normal information already present. As a result, normals are generated for every triangle and vertices of the input geometry in order to sign those unsigned voxels of the Cartesian grid column that are touching the embedded geometry. Figure 9 shows the position of solution columns for the Stanford Bunny [57]. As illustrated in the figure, these solution columns do not intersect and surround the holes on the surface of the geometry.

Numerical Solution of the Diffusion Equations

Diffusion is a well-known and well-understood time-dependent process, constituted by random motion of given entities and causing the statistical distribution of these entities to spread in space. The diffusion equation solution space can be defined as a set of Cartesian grids embedding curved thin plates in 3D space where one side is heated while the other side is cold, diffusing energy through 3D space. The curved plates embedded in the Cartesian grids represent segments of the input model embedded in *solution columns*. The equations derived in Appendix A are used to find a steady-state solution for the diffusion equation. A finite-volume formulation for the diffusion equation is employed to arrive at a solution for the diffusion equation as described in equation (15). Equation (17) provides the discretization of the finite-volume formulation. It is an explicit scheme with a forward difference in time and central difference in space. Equation (18) provides the solution of the diffusion equation.

The diffusion equation is being solved as a time dependent problem, and the error introduced at any time step is going to grow or decay based on the stability condition. Von-Neumann's analysis predicts that the conditions for the numerical scheme to be stable. The stability analysis provides a range of values of $\alpha\Delta t$ for which the numerical scheme used is stable. The acceptable values of $\alpha\Delta t$ depend upon the grid refinement, as given in equation (32) for Cartesian Grids. Equations (34), (35) and (36), as defined earlier, provide three measures of change to determine whether or not the solution has reached convergence. $\%\ L2Norm_{change}^{n+1}$, as defined in equation (36), is the average percentage change for a Cartesian grid compared to the previous iterations and is used in most of this study to determine the convergence of the solution of the diffusion equations. Two parameters, $\%\ L2Norm_{change}^{n+1}$ and the number of iterations, are used to determine whether or not convergence has been reached. The cutoff values of these parameters, 0.5% or 1000 iterations, respectively, are based on empirical evidence from experimental results. Figure 10 and Figure 11 show the convergence plots of L_2Norm_{Max} and L_2Norm_{Avg} for Columns 2 and 3, respectively, for the Stanford Bunny. In both the cases, the plots are asymptotic and seem to converge within a few iterations. However, looking closely at the log plots in Figure 12 and Figure 13, it can be inferred that convergence is actually reached much later.

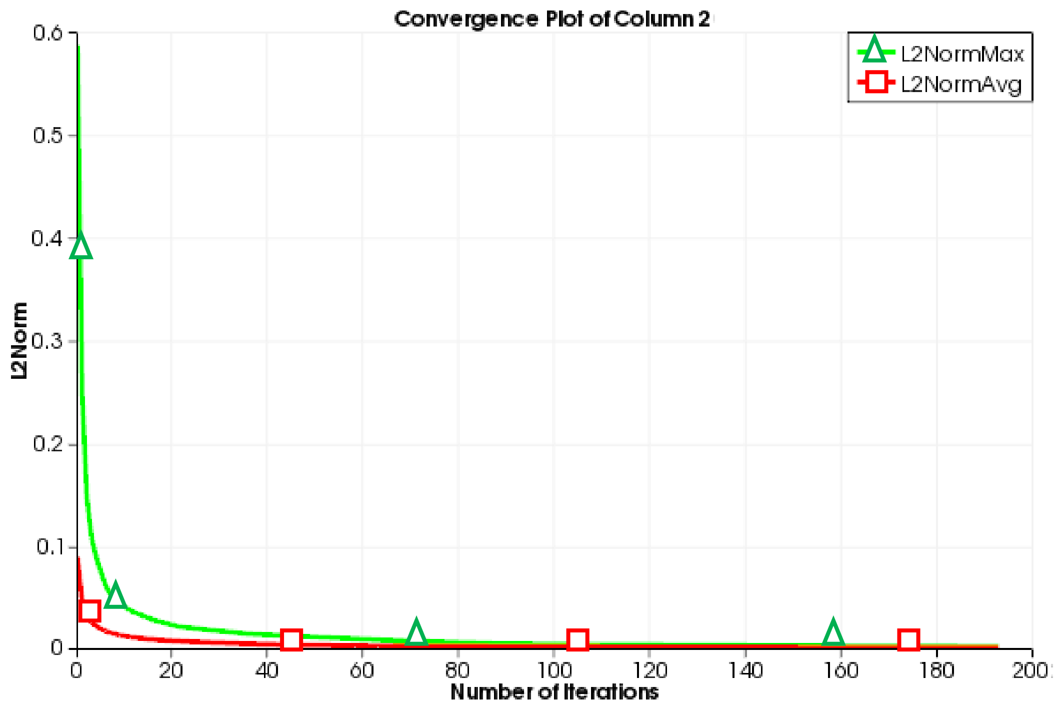


Figure 10: Convergence Plot for Column 2 of the Stanford Bunny

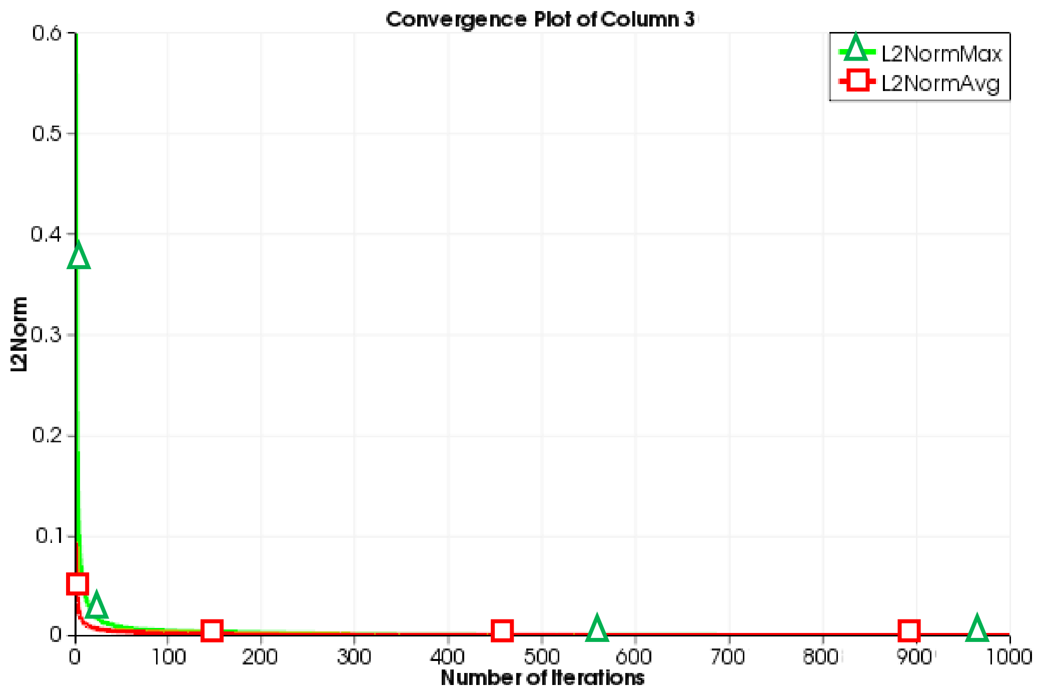


Figure 11: Convergence Plot for Column 3 of the Stanford Bunny

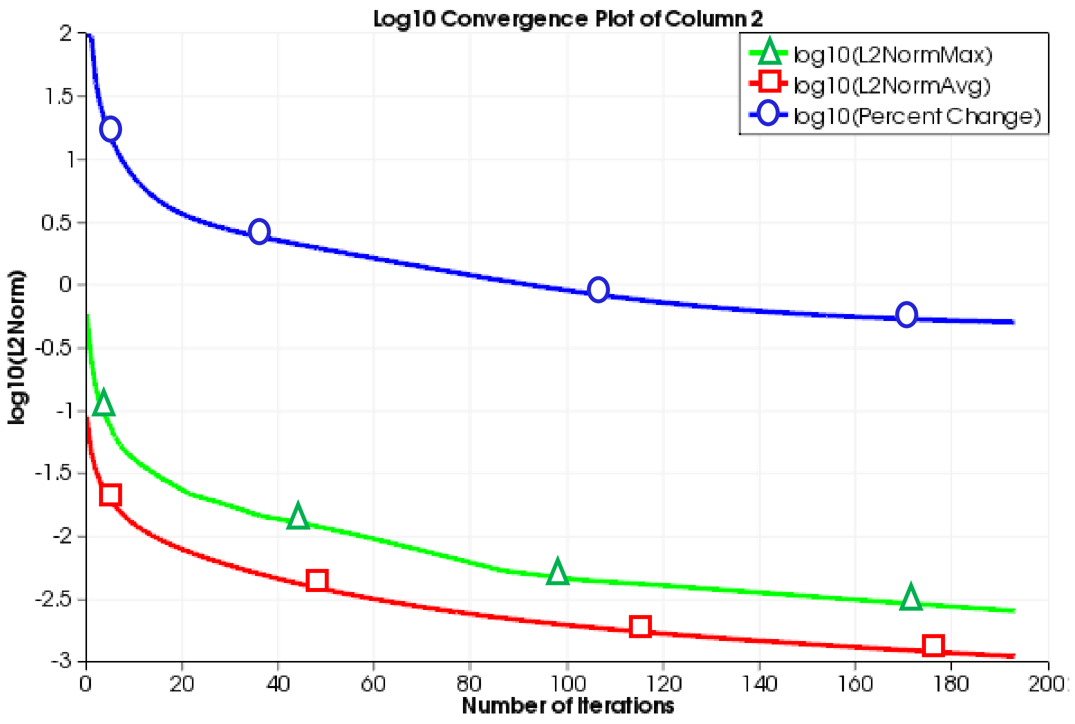


Figure 12: $\log_{10}$ Convergence Plot for Column 2 of the Stanford Bunny

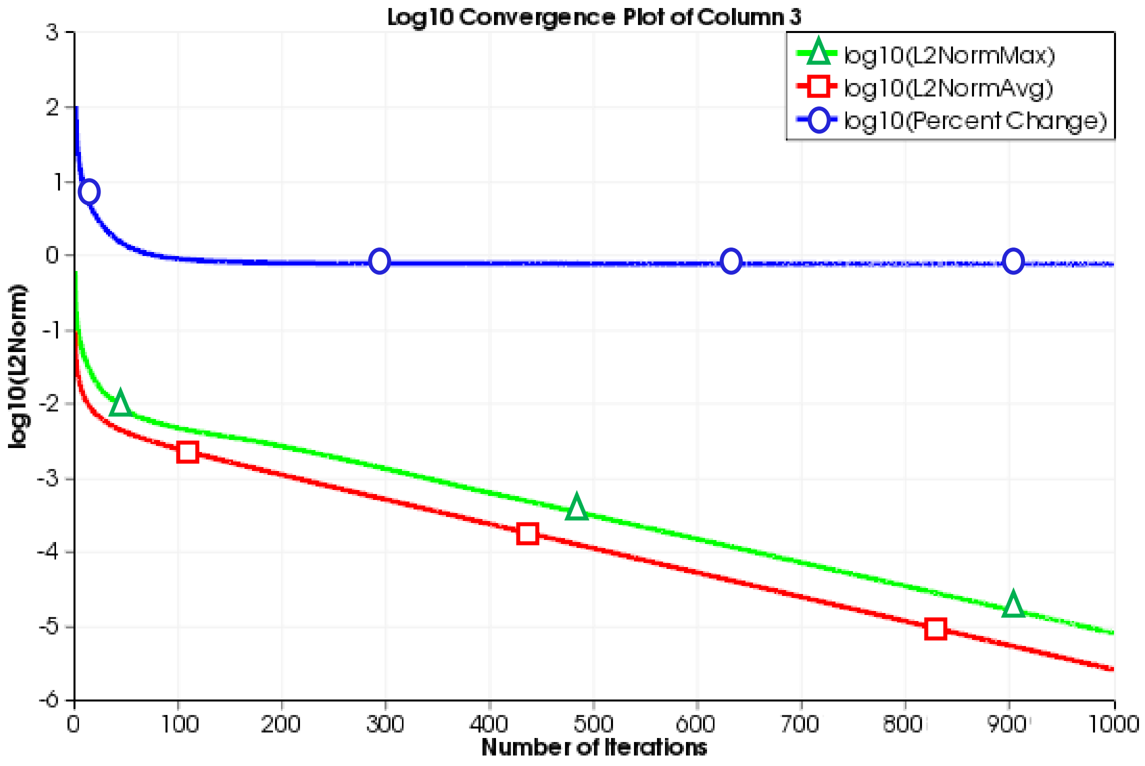


Figure 13: $\log_{10}$ Convergence Plot for Column 3 of the Stanford Bunny

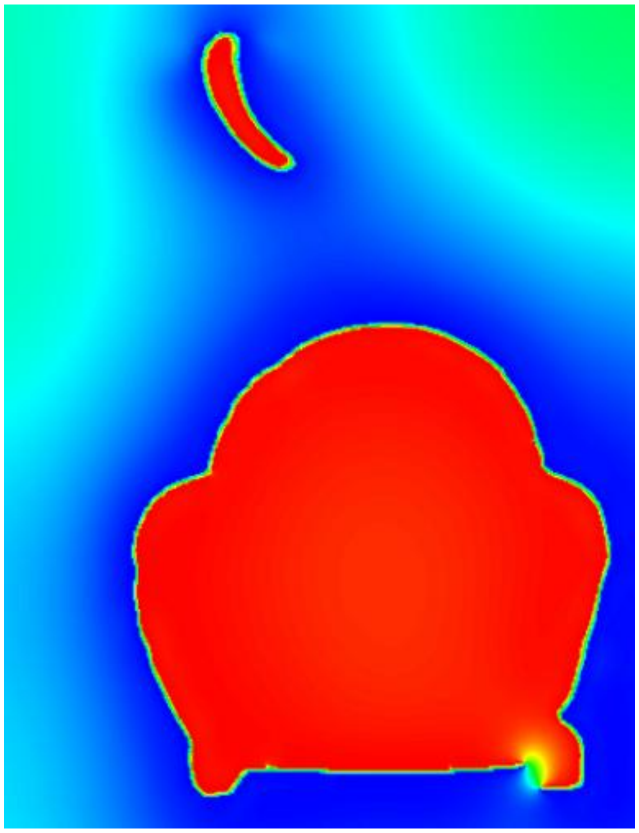


Figure 14: Rendering of a slice of the Stanford Bunny in false color

Column 2 is a small Cartesian Grid, and the diffusion equation solver reaches convergence much faster than when the same diffusion equation solver is run on Column 3, as shown by Figure 10 and Figure 11.

Explicit schemes are known to be notoriously slow for convergence. The convergence becomes even slower as one refines the grid to a finer resolution. To circumvent this problem, APIs, which can be used for a primitive algebraic interpolation approach using a coarse grid to initialize the flow field for the finer Cartesian grid, have been implemented. This allows the overall solution field to be initialized using the preliminary solution from the coarser grid, which can reduce the time needed for convergence. First the solver is run on a coarse grid for a fixed number of iterations. The solution from the coarse grid is interpolated onto the finer Cartesian grid as initial values on which the diffusion solution process is started for better convergence.

The diffusion equation provides a smooth and continuous variation of the solution at the time of convergence. It is a Laplace equation that satisfies Min-Max property, which states that the computed values inside the solution domain would lie between the minimum and maximum values specified at the boundary when the solution has converged. This ensures that the computed values in the domain will lie between '-1' and '+1' in the simulation of the equation. These two properties make heat equation a good choice for the problem being solved in this research, where one wants a smooth and continuous variation of the solution at the time of convergence, resulting in a smooth and continuous surface as output after contouring.

Figure 14 shows the rendering for a slice of Stanford Bunny in false color after convergence on GPU when the whole model is embedded in one Cartesian grid column for illustration purpose. The rendering is done using Paraview [64] for a slice of a fine Cartesian grid with 420 voxels along the maximum dimension, which was initialized with the solution from a coarse Cartesian grid with 100 voxels along the maximum dimension. One can clearly see a diffusion of colors in the lower right corner of the rendering, signifying a discontinuity in the model and a gradient in the temperature values in that area due to diffusion. The diffusion solution in this case was achieved using CUDA on a Quadro FX 5800 GPU donated by NVIDIA for this study.

Extraction of Consistently Oriented Surfaces and Point Set

The zero-set of the numerical solution for the diffusion equation, as described in the previous section, provides a closed surface. This zero-set surface is otherwise only open at the places where it intersects with the extremities of the Cartesian grid. Extraction of the surface using a suitable contouring method should provide us with the desired result. There are a number of contouring methods available, the most common of which is Marching Cubes [40]. The Marching Cubes algorithm is feature insensitive and introduces aliasing artifacts in the form of a *staircasing* pattern on the surface while greatly increasing the number of triangles on the surface, depending on the resolution of the Cartesian grid, often giving poor results. A number of open-source implementations of the Marching Cubes algorithm are widely available in the public domain. In this implementation, a VTK library implementation of the Marching Cubes algorithm specialized for 3D image data, known as `vtkImageMarchingCubes` [61], is used. It was chosen due to its open source availability and ease of use. The extracted surface can contain aliasing artifacts and can also contain topologically complex artifacts in the form of small bubbles attached to the extracted surface. These artifacts can be easily removed by first checking the connectivity of the output surfaces and by extracting the largest surface. Next the extracted surface mesh is relaxed using a smoothing filter, such as the one described by Taubin et al. [54]. `vtkWindowedSincPolyDataFilter` [62] is an open-source implementation of this smoothing filter and is available as part of the VTK library. Figure 15 shows the extracted surface on a *solution column* after smoothing.

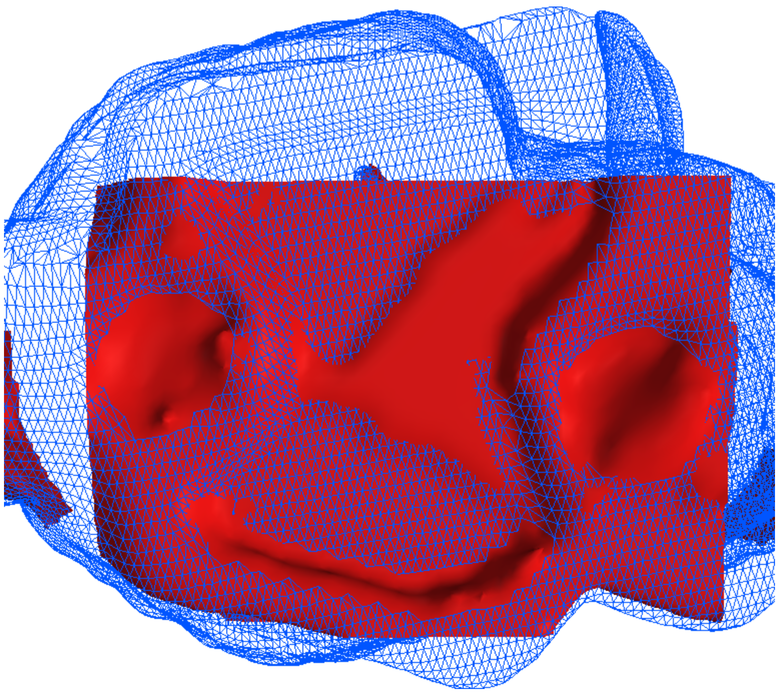


Figure 15: Contouring and smoothing result on a column at the base of the Stanford Bunny

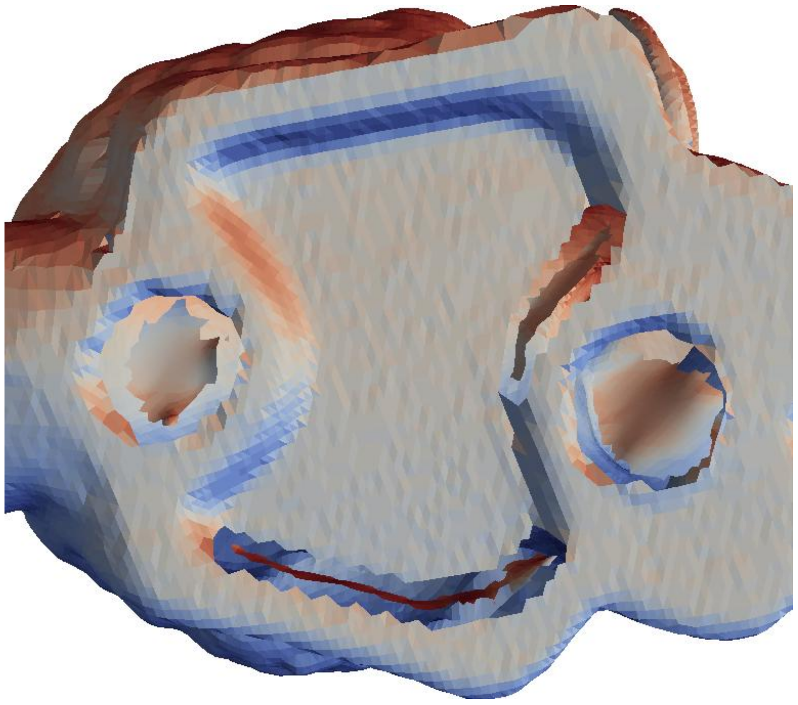


Figure 16: Extracted patch from the contouring result superimposed with the original model.

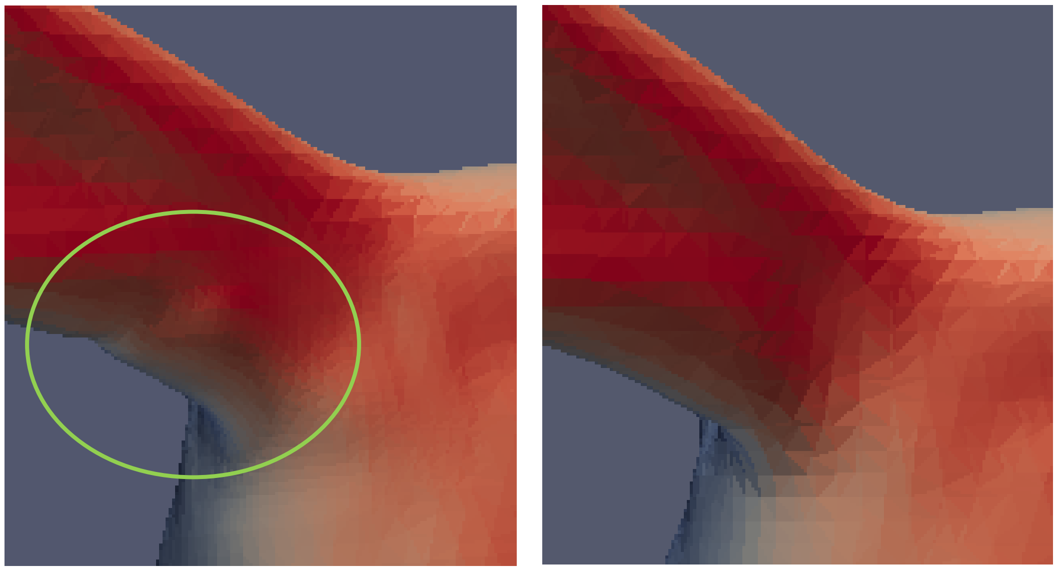


Figure 17: Reconstructed Surface (a) with bump; (b) without bump

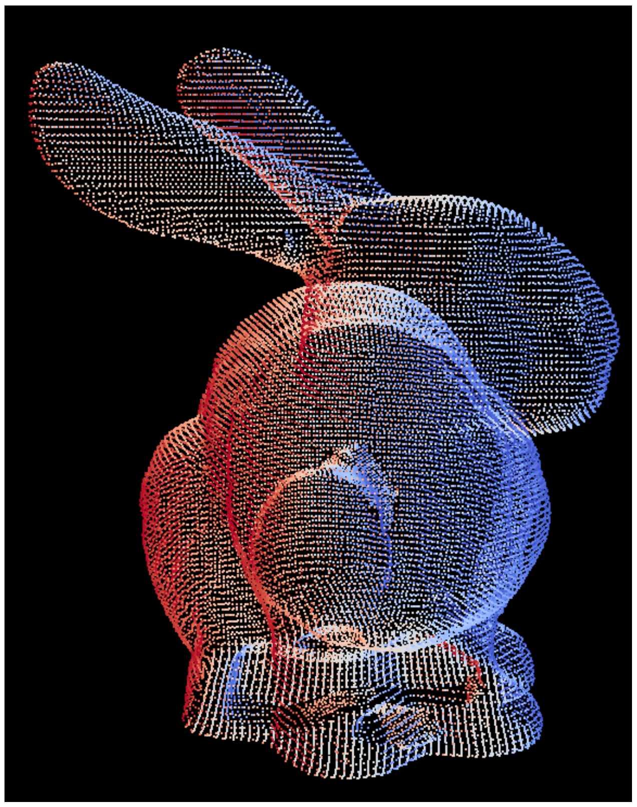


Figure 18: Extracted point set with consistently oriented normals

Once the contoured and smoothed surface is extracted, all triangles that intersect with the voxels embedding the original surface are removed from the extracted surface. This process provides us with an extracted patch in the hole region. Figure 16 shows extracted patches superimposed with the original input mesh with holes. By careful study, it was observed that the conversion of a surface mesh to an intermediate volume mesh and subsequent contouring causes a shift of about half a voxel in the position of the extracted surface when compared with the original surface. It is believed that this happens because the original surface is embedded in voxels, while contouring is done by evaluating values on the voxel corner. If left uncorrected, this would cause a noticeable shift in the reconstructed surface, as evidenced in Figure 17 (a). It is corrected by shifting the extracted surface by 0.5 voxels along inward normal. This correction resolves the bump to get a smoother surface after reconstruction in the region where the patch and holes lie, as shown by Figure 17 (b). The normal information in the original and reconstructed surface may not be present *a priori* and, as a result, the normal at every vertex of the input surface and extracted patch has to be calculated.

All vertices on the input surface mesh along with isles and patches are extracted and normals are generated to create a well-sampled point set, as shown in Figure 18. The point set shown in Figure 18 has been rendered based on the normals generated on those points. This well-sampled point set would be used for surface reconstruction using the Poisson Surface Reconstruction technique [38] as described in the next section.

Surface Reconstruction and Results

Surface reconstruction and surface fitting from point samples is a well-studied problem in computer graphics and has applications in a number of disciplines, including surface reconstruction from input points and reverse engineering. Reconstruction itself is a very challenging area due to uneven sampling of points, noisy data and scan mis-registration, among other problems. There are a number of schemes for surface reconstruction based on implicit forms among other techniques. The implicit surface fitting methods are either global or local in nature. Global fitting methods commonly define the implicit function as the sum of radial basis functions (RBFs) centered at the points. Local fitting methods consider subsets of nearby points. These methods are well studied and have been compared in a number of papers including, but not limited to, Amenta and Bern [28], Amenta et al. [29], Carr et al. [30], Bruno [31], Dey and Goswami [32], Dey and Goswami [33], Shen et al. [34], Casciola et al. [35], Kazhdan et al. [36] and Mullen et al. [37], among others.

In this research, an existing and well-established surface reconstruction method was used, which is Poisson Surface Reconstruction as described in Kazhdan et al. [36]. It is an open-source algorithms and is widely available [38]. Although Poisson surface reconstruction provides a watertight surface, its accuracy depends on the sampling of input points. Therefore, in this study, efforts are made to provide a well-sampled point set with correctly oriented normals as input to the Poisson surface reconstruction in order to get an output surface that is not only smooth but also well behaved. The point set generated for the input surfaces, as well as the extracted patches and surface reconstruction creates a completely new reconstructed watertight surface for the whole model, which may be quite different from the input surface. The VTK implementation of Poisson surface reconstruction as presented by Doria and Gelas [38] is used at Octree refinement level 10 in this study wherever Poisson surface reconstruction is used to reconstruct a repaired model. Figure 19 shows the reconstructed watertight model of the Stanford Bunny, which was generated using the Poisson surface reconstruction as the final step of a volume-based mesh repair. The point set shown in Figure 18 was used as an input during surface reconstruction. The surface generated after using the Poisson surface reconstruction is the final result obtained from the volume-based mesh repair algorithm presented in this chapter, as shown by Figure 19.

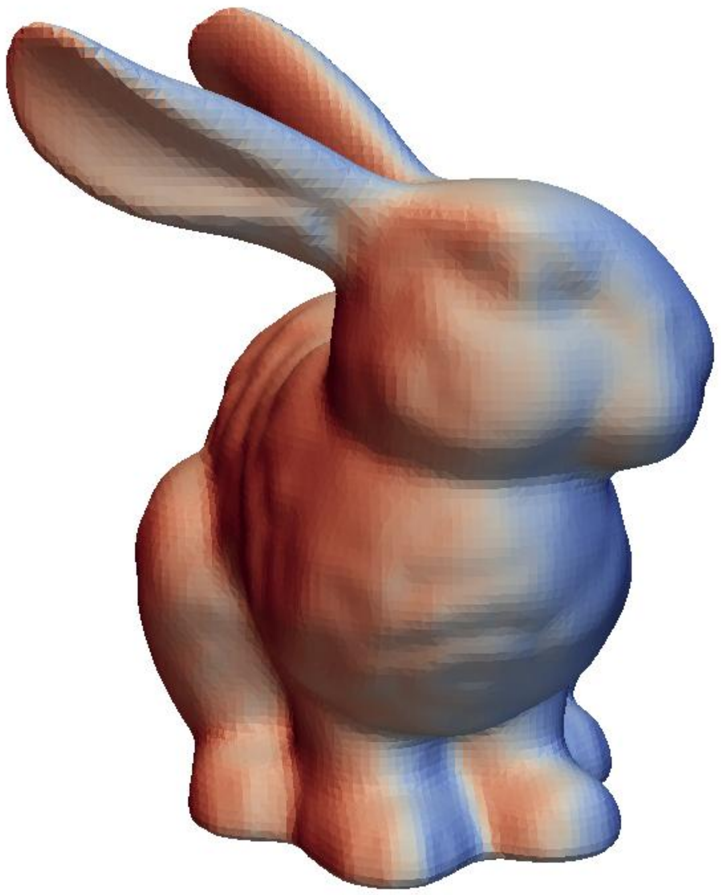


Figure 19: Reconstructed watertight model of Stanford Bunny

In this chapter, a volume-based approach is presented that can repair a discrete input geometry by solving the diffusion equation followed by a Poisson surface reconstruction. Even though the volume-based method is applied on isolated *solution columns,* the repaired and reconstructed watertight model after Poisson reconstruction is altered from the original input model during the reconstruction process. Volume-based approaches like the one described in this chapter can be used to repair the models with artifacts that surface-based models otherwise cannot robustly handle. However, they also pose some potential problems. The conversion to and from an intermediate volume representation leads to the resampling of the model. It may significantly alter the input geometry, which could result in the loss of model features and could destroy any structure that might have been present in the connectivity of the input model. Despite all their shortcomings, volume-based algorithms can solve some problems in mesh repair robustly that cannot be handled by surface-based approaches alone.

Mesh Repair Results on Analytical Models

In order to evaluate the accuracy of this volume-based hole-patching algorithm, ellipsoids are chosen as benchmark cases to calculate the average errors in terms of radius and the standard deviation of errors of the location of each point. This was done due to the fact that the location of each point on an ellipsoid can be analytically defined. For comparison purpose, results from surface-based technique is also presented in Table 1.

An ellipsoid with semi-axes *a*, *b*, and *c* and centered at coordinate (0, 0, 0) is defined as:

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}=1 \tag{1}$$

Error in the location of points on the ellipsoids can be quantified as:

$$E_i=\frac{{x_i}^2}{a^2}+\frac{{y_i}^2}{b^2}+\frac{{z_i}^2}{c^2}-1 \tag{2}$$

The average error can be measured as:

$$\overline{E}=\frac{\sum_i^N E_i}{N} \tag{3}$$

The standard deviation is the measure of the spread in a set of values. The standard deviation in the error of coordinate positions on the patches can be obtained as follows:

$$\sigma=\sqrt{\frac{\sum_i^N\left(\overline{E}-E_i\right)^2}{N-1}} \tag{4}$$

If the minimum distance by which the position of a point (x_i, y_i, z_i) on the ellipsoidal surface could be moved to a new location (λx_i, λy_i, λz_i) such that the point at the new location would lie exactly on the surface of the analytical ellipsoid, then the value of λ can be defined as:

$$\lambda=\pm\sqrt{1\bigg/\left(\frac{{x_i}^2}{a^2}+\frac{{y_i}^2}{b^2}+\frac{{z_i}^2}{c^2}\right)} \tag{5}$$

As a result, the displacement error % in the position of the point (x_i, y_i, z_i) can be given as

$$\text{Displacement error \% in position} = \left(\frac{1}{\lambda} - 1\right) * 100 \tag{6}$$

Tables 1 and 2 present the errors in the position of the points of the reconstructed model using surface-based and volume-based repair techniques, respectively. The tables show the number of points, average error, standard deviation of error, minimum absolute displacement error % and maximum absolute displacement error % due to the the position of points on the reconstructed surface of three ellipsoids. The three ellipsoids were constructed by varying semi-axis *b* along *y*-direction. A sphere is a special case of an ellipsoid when *a* = *b* = *c*. It can be observed from tables 1 and 2 that geometry generated using surface-based mesh repair technique provide better results when compared with that of the volume-based mesh repair technique presented in this chapter. This is an expected outcome. It has been empirically observed that the errors due to the location of the points on the output geometry using volume-based repair tend to become smaller as the resolution of the input model is increased.

Model Name	Number of Points on Surface	Average Error	Standard Deviation of Error	Minimum Absolute Displacement Error %	Maximum Absolute Displacement Error %
Sphere (a = 1.0, b = 1.0, c = 1.0)	30413	-5.415e-04	1.748e-03	0.0	0.688
Ellipsoid (a = 1.0, b = 0.5, c = 1.0)	30514	-4.942e-04	1.778e-03	0.0	0.850
Ellipsoid (a = 1.0, b = 0.2, c = 1.0)	30403	-5.452e-04	2.673e-03	0.0	2.188

Table 1: Result of surface-based mesh repair technique on analytical models

Model Name	Number of Points on Surface	Average Error	Standard Deviation of Error	Minimum Absolute Displacement Error %	Maximum Absolute Displacement Error %
Sphere (a = 1.0, b = 1.0, c = 1.0)	32680	1.347e-02	0.02908	2.282 -06	7.381
Ellipsoid (a = 1.0, b = 0.5, c = 1.0)	36099	2.364e-02	0.05957	3.080e-06	16.442
Ellipsoid (a = 1.0, b = 0.2, c = 1.0)	34298	7.708e-02	0.21952	1.559e-05	50.729

Table 2: Result of volume-based mesh repair technique on analytical models

CHAPTER 5

HYBRID APPROACH TO GEOMETRIC HOLE PATCHING

Surface-based methods explicitly try to identify surface artifacts prior to repairing them and require that input geometry meets some mesh quality conditions. This is sometimes not feasible, causing the mesh repair techniques to fail. However, the surface-based methods have some compelling advantages over the volume-based methods, which emphasize that surface-based techniques should be employed wherever possible to get better quality results compared to using only volume-based methods for geometry repair.

The volume-based approach presented in chapter 4 works well for relatively simple input surfaces but will not provide well-behaved and feature-sensitive results for inputs having high curvature and complicated geometries. In those cases, a hybrid approach can be used. A hybrid approach leverages the benefits of both surface-based and volume-based approaches to obtain a reconstructed watertight surface, which is not only smooth and well-behaved, but also feature-sensitive.

A hybrid approach would be used when the surface-based algorithm has failed to repair the input geometry completely. All the generated patches from the surface-based approach, along with the input geometry, are used as inputs to the volume-based approach described in chapter 4. Figure 20 presents the flow chart for the hybrid approach to mesh repair presented in this chapter. In the hybrid approach to geometry repair, parts of the original input geometry, along with all the surface fragments as well as the output surface patches from the surface-based approach, are embedded in the voxelized, non-intersecting Cartesian grid *solution columns* prior to generating a diffusion equation-based solution in those columns, as described in chapter 4.

As part of a previous effort, an automatic surface-based method has been presented and published for patching topologically simple holes on a triangulated surface model to achieve a watertight surface, as described by Kumar et al. [12], Kumar and Shih [13] and Kumar et al. [14]. The existing surface points around the holes were used to obtain a set of NURBS surfaces approximating the missing surface patches. A Delaunay triangulation method and repeated point insertions at the centroid of the triangles were used to generate internal points that were then projected onto a set of NURBS surfaces to obtain the desired patch. The patches generated by this method were achieved with minimal alteration of the geometric information of the surrounding geometry. This algorithm for surface-based mesh repair is applicable to topologically simple but geometrically complex holes in the discrete geometry. Such holes are common in the geometries obtained from 3D scanners or extracted from medical image datasets using the Marching Cubes algorithm. This hybrid approach uses an incremental improvement to the surface-based approach for mesh repair. This insures that the input to the volume-based algorithm in the hybrid approach is a surface that shows better feature sensitivity than what would have been available without using any surface-based approach.

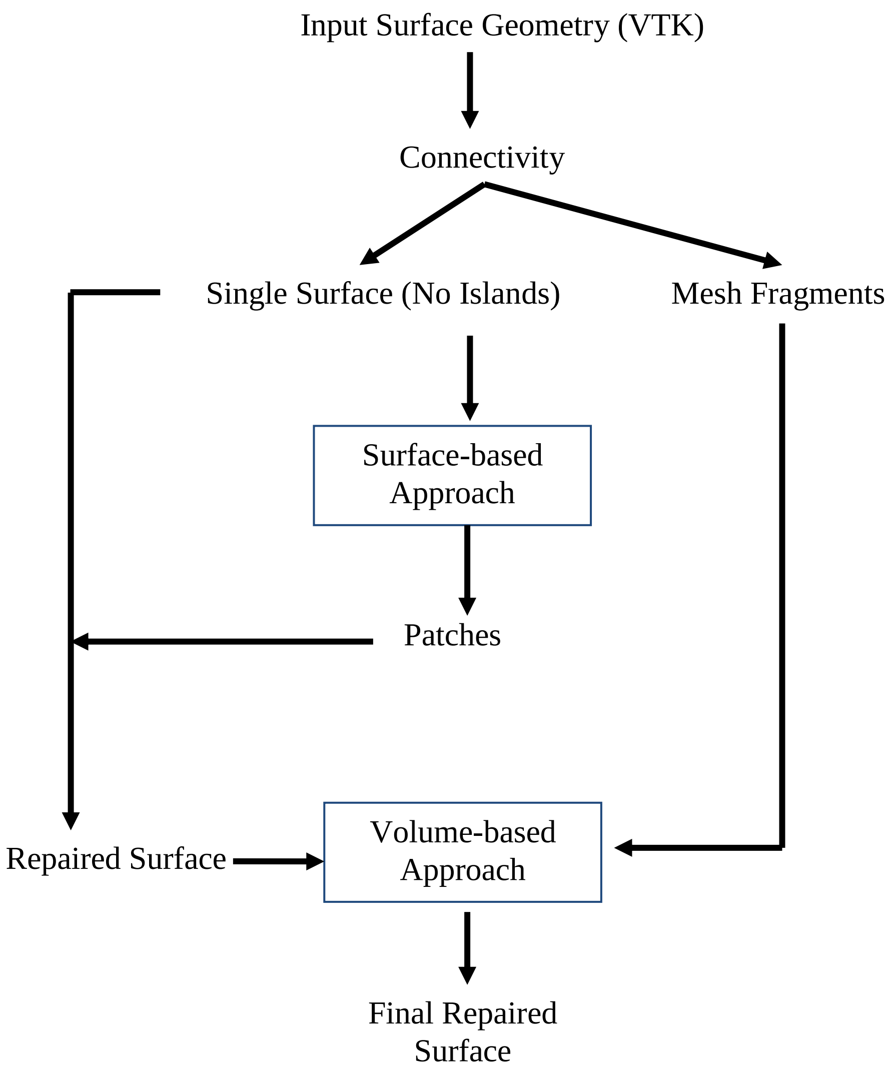


Figure 20: Flowchart of hybrid approach towards mesh repair.

The main features of this surface-based mesh repair algorithm, along with a number of improvements which have been implemented and used in the hybrid approach presented in this chapter, are as follows:

- No assumption about the orientation, shape, size or origin of the holes on the surface.
- Analogy of non-intersecting rings on unstructured mesh surrounding gaps or holes on the surface mesh.
- Fully automatic method with the size and density of the triangulation in the hole patching process using incremental refinement controlled by the size of neighborhood triangles (edge length and area).
- Point insertion at the centroid and edge swapping based on Delaunay criteria, as described in Appendix C.
- Smooth patches even in the regions with high curvature.
- Explicit identification of holes and their sorting based on their sizes in terms of number of edges.
- Octree-based search for locating points and a hybrid octree search for location of edges and triangles on the mesh.
- Hole patching using localized NURBS-based surface definition.
- Hole patching only supported with discrete geometries with simple topologies. The presence of isles in the hole region is detected but not supported.

The algorithm for this hybrid approach, as described in the flowchart shown in Figure 20, will be validated and presented with examples using the Laurent Hand [58] and Chinese Lion [59] models. Similar to the Stanford Bunny model, both of these models are freely available online in high resolution and are used widely for research purposes for validation of algorithms. These models are highly complex and lack water-tightness. Some of the holes present on the surfaces of these models are in areas with high geometric curvature in narrow areas and pose a significant challenge for geometry repair using traditional methods. The surface-based algorithm that would be used on these models failed to completely repair the mesh, while the volume-based algorithm presented in chapter 4 failed to produce a satisfactory result on these models due to the close geometric proximity of finger surfaces. The hybrid approach presented in this chapter is not only able to repair the models but to produce results that conform to what are expected based on the surrounding features. These cases are discussed and illustrated below.

Laurent Hand:

The Laurent Hand model [58] was scanned at INRIA by Laurent et al. and is available at aim@shape shape repository [63]. It is a non-manifold model and lacks water-tightness, with a number of gaps in geometrically complex areas due to occlusion as shown in Figure 21. The next few figures show the results of different techniques on this model. The region surrounding three middle fingers is illustrated in Figure 22. The surface between the thumb and index fingers has high curvature and are in close proximity. Applying the volume-based technique in this area without any extra data, such as line of sight information, would end up fusing the fingers together, as demonstrated in Figure 25 (b) and (c).

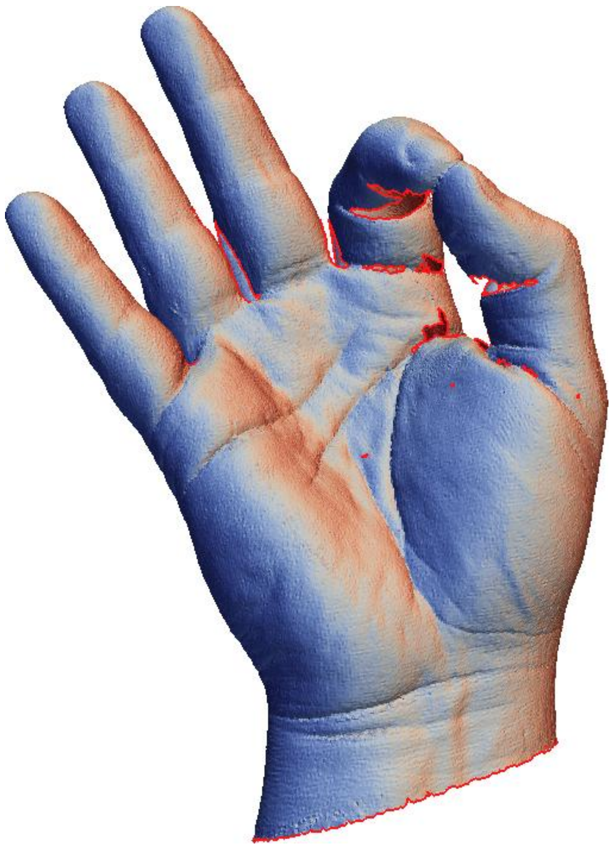


Figure 21: Laurent Hand model with a number of discontinuities

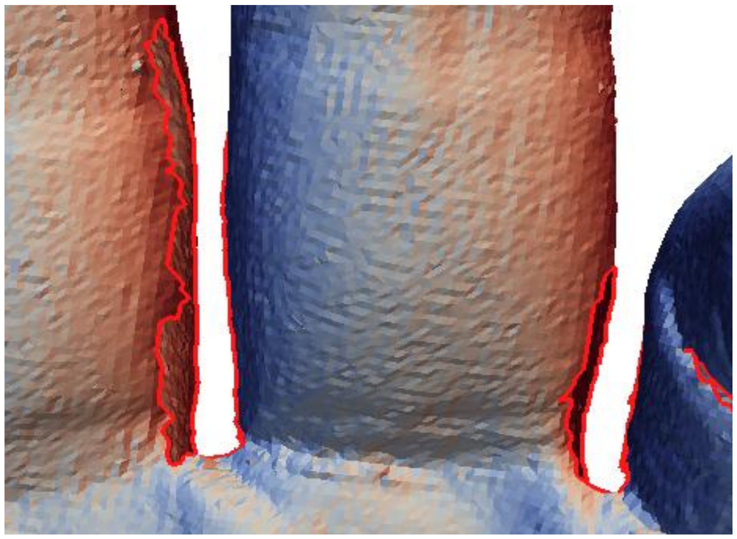
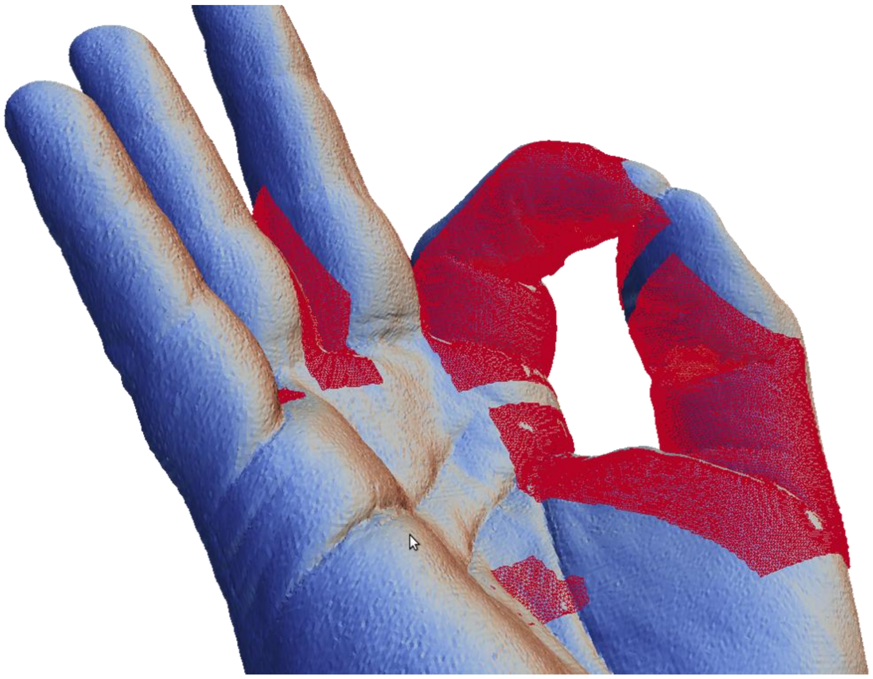
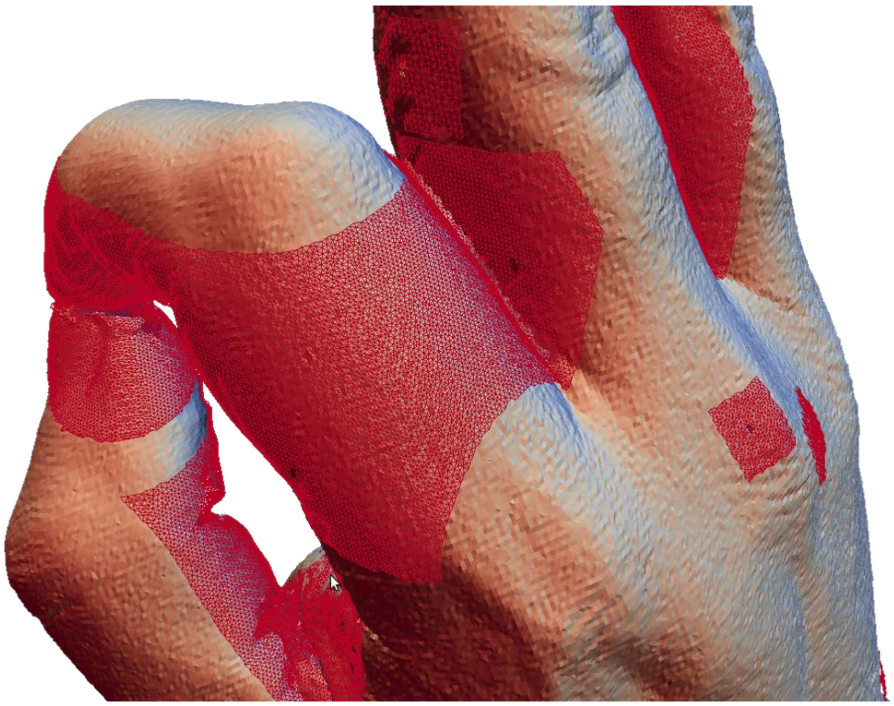


Figure 22: Discontinuities in Laurent Hand model around three middle fingers

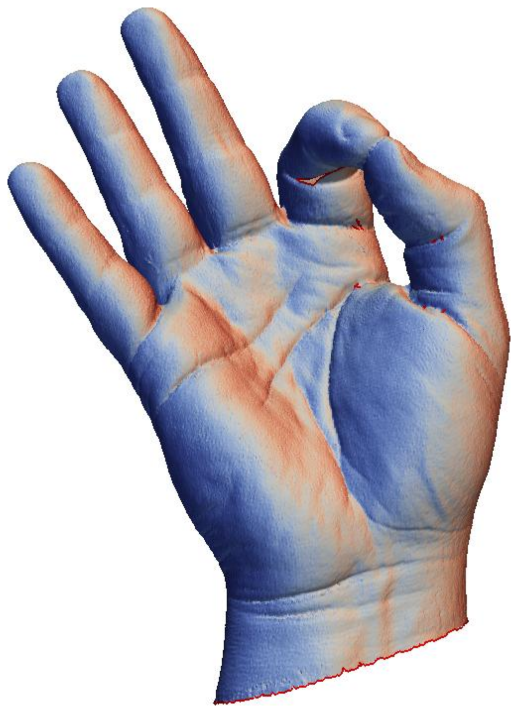


(a)

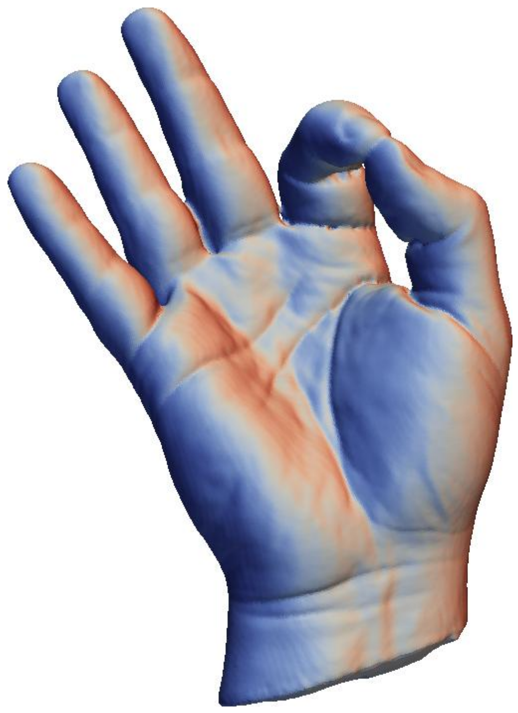


(b)

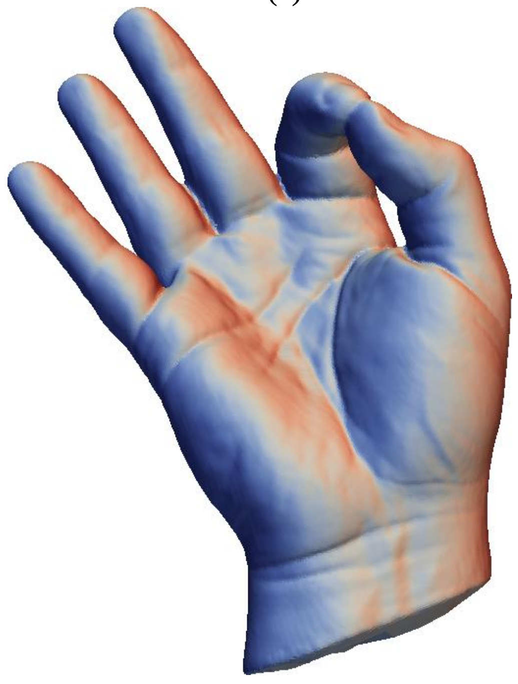
Figure 23: Extracted surfaces after contouring super imposed on the original Laurent Hand model showing (a) front (b) back



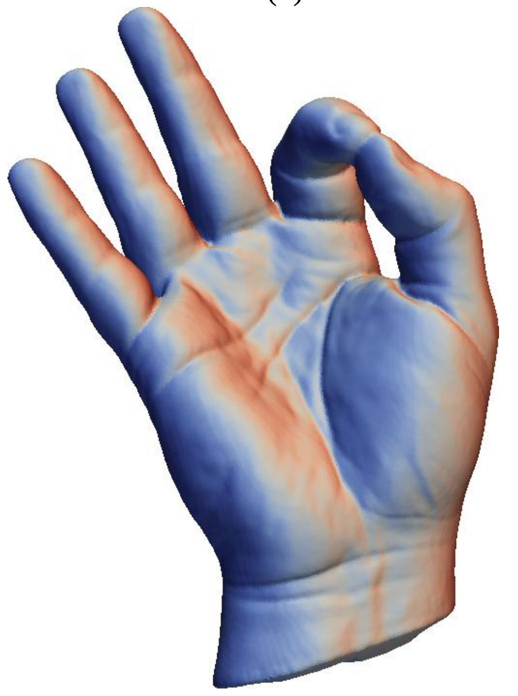
(a)



(b)



(c)



(d)

Figure 24: Laurent hand after repair: (a) surface-based; (b) volume-based with reconstruction; (c) with only Poisson reconstruction; (d) with hybrid approach

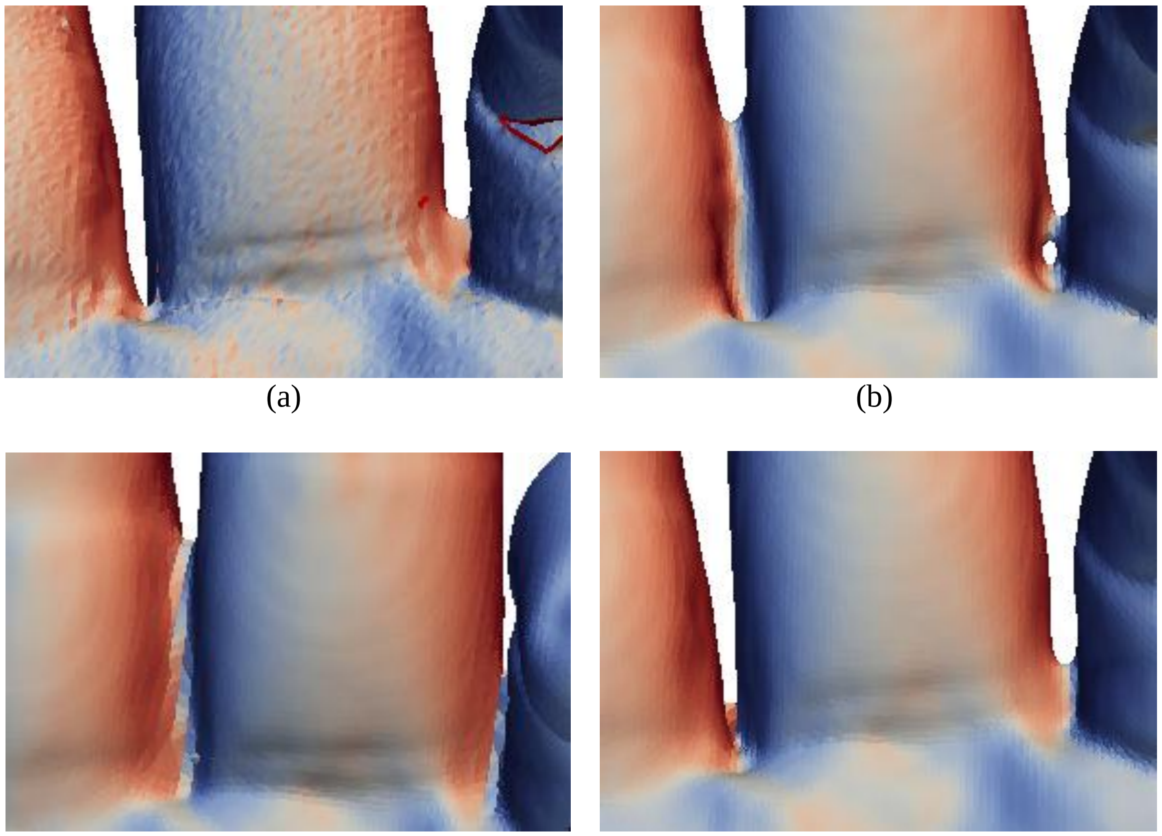
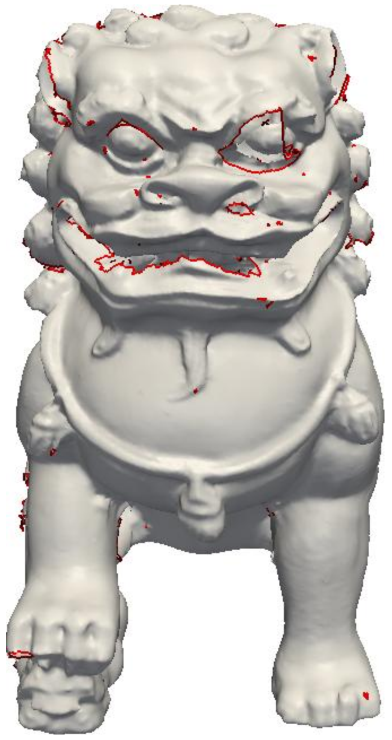

(a) (b) (c) (d)

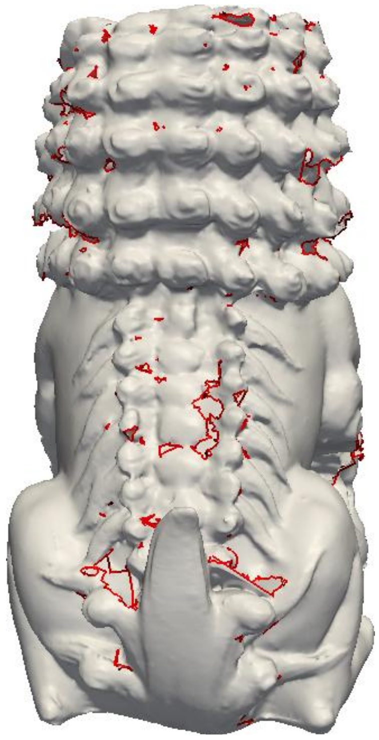
Figure 25: Middle three fingers in Laurent hand after repair: (a) surface-based; (b) volume-based with reconstruction; (c) with only Poisson reconstruction; (d) with hybrid approach

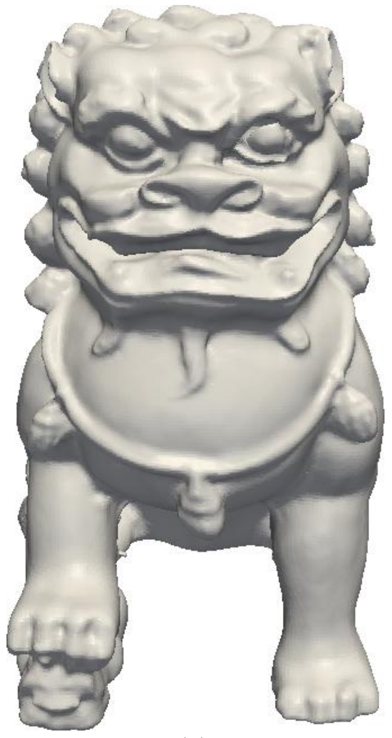
Figure 23 (a) and (b) show non-intersecting surfaces extracted from *solution columns* after contouring and their placement on the original model. Figure 24 (a) shows mesh repair on the Laurent Hand using the surface-based approach. It can be observed from Figure 24 (a) and (b) that although the surface-based approach is successful in repairing the mesh between the fingers, it is not able to close all the voids. Figure 24 (b) shows a model repaired using the volume-based approach, as described in this chapter which uses Poisson surface reconstruction as the final step. It can be observed that, although the output geometry is watertight, the output has fused fingers, as shown in Figure 25 (b), which is not desirable. Figure 24 (c) and Figure 25 (c) show the output for only Poisson-based surface reconstruction, and it can be observed that, although the output geometry is watertight, it has all three middle fingers fused together, which is again undesirable.

Figure 24 (d) and Figure 25 (d) represent output from the hybrid approach, which uses both surface-based and volume-based techniques followed by Poisson surface reconstruction. It can be observed that not only is the output surface mesh watertight, but the middle fingers in the narrow area are not fused, as desired, compared to the cases when only a surface- or a volume-based technique was used to repair the geometry. This example demonstrates the usefulness of hybrid technique for repairing a surface geometry.

(a) (b)


(c)


(d)


(e)

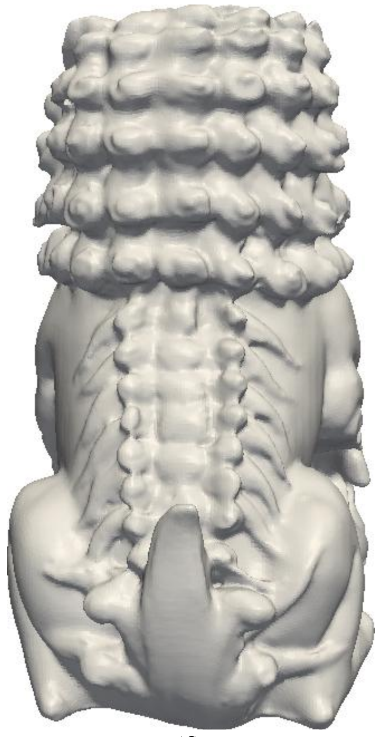

(f)

Figure 26: INRIA Chinese Lion model: (a) original, front; (b) original, back; (c) surface-based repair, front; (d) surface-based repair, back; (e) hybrid approach, front; (f) hybrid approach, back

INRIA Chinese Lion

The INRIA Chinese Lion [59] model was scanned by the Geometrica Group at INRIA using the Minolta Vivid 910 Laser Scanner. The model is available on the Aim@Shape Project Shape Repository [63]. The Chinese Lion model so obtained is a dirty model with non-manifold complex edges and numerous small surface fragments and defects. Not all holes could be filled using the surface-based approach [12], [13], [14] due to the complexities of the model, as well as inherent inadequacies of the initial surface triangulation algorithm in 3D space. Figure 26 (a) – (f) demonstrate the result of geometry repair on the model using the hybrid approach and its visual comparison with the original model as well as the model generated after failed surface-based mesh repair on the model. Figure 26 (a) and (b) show the front and back rendering of the original model along with numerous holes on the surface. It can be observed that a number of those holes are geometrically very complex. Figure 26 (c) and (d) show the front and back rendering of the Chinese Lion model after surface-based mesh repair. One can notice that a number of the holes and discontinuities presented in Figure 26 (a) and (b) have been either completely filled or have contracted in size. However, surface-based mesh repair is not able to fill all the holes. Figure 26 (e) and (f) show the final result of the mesh repair process using the hybrid approach that uses output from the surface-based approach, as shown in Figure 26 (c) and (d) along with isles and surface fragments present in the original model as input to the volume-based repair process. The resultant output mesh is completely watertight.

CHAPTER 6

DIFFUSION SOLVER PARALLELIZATION

With the development of increasingly complex technologies, there is a continuous demand for greater computational power (speed, memory, and bandwidth) than what is presently possible. In particular, realistic numerical simulations in scientific and engineering problems needs ever-increasing computation resources, since such simulations often require a large number of repetitive calculations on a relatively large amount of data in a limited time in order to generate valid, useful results. Traditionally, such problems were solved in a parallelized fashion on specialized machines known as supercomputers. These supercomputing machines were housed in large buildings with restricted access and needed large investments in terms of dedicated staff, special cooling machines, specialized proprietary interconnect hardware and memory switches. These days, with advancement of computer technologies, necessary computing resources such as computing power, memory and bandwidth is becoming increasingly more cost effective and smaller in hardware size. As a result, it has become possible to obtain the kind of computing power through commodity multiprocessors, multi-core desktop computer systems and off-the-shelf graphics cards that would have required access to specialized supercomputers only a few years ago.

A multiprocessor system is a parallel computation machine that contains more than one processor. Multiprocessor computation includes multi-threaded computations, which could run concurrently on a single machine with multiple processors or processor cores. The speedup factor $S(p)$, or speedup, which is a measure of relative performance of such a multiprocessor system, can be defined as

$$S(p) = \frac{Execution\ time\ on\ single\ processor}{Execution\ time\ using\ a\ multprocessor\ with\ p\ processors} \tag{7}$$

The maximum possible theoretical speedup is usually *p* with *p* processors when the problem can be equally divided among the *p* processor with no additional overhead costs, as shown in equation (8) below.

$$S(p) = \frac{t_s}{t_p} \leq \frac{t_s}{t_s/p} = p \tag{8}$$

Most of the computations have additional fixed overhead of job and thread scheduling along with communication cost between processes, which scales with the size of the problem. The maximum possible speedup for such problems of fixed size is defined by Amdahl's Law [65] .

Amdahl's law assumes that every parallel computation has a serial part that can only be executed on one processor. If the fraction of computation that cannot be divided into concurrent tasks is *f*, then the speedup according to Amdahl's law can be given as

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{p}{1+(p-1)f} \tag{9}$$

Amdahl's law puts a maximum upper limit for speedup on a problem of fixed size with an increasing number of processors, as shown below.

$$\underset{p \to \infty}{S(p)} = \frac{1}{f} \tag{10}$$

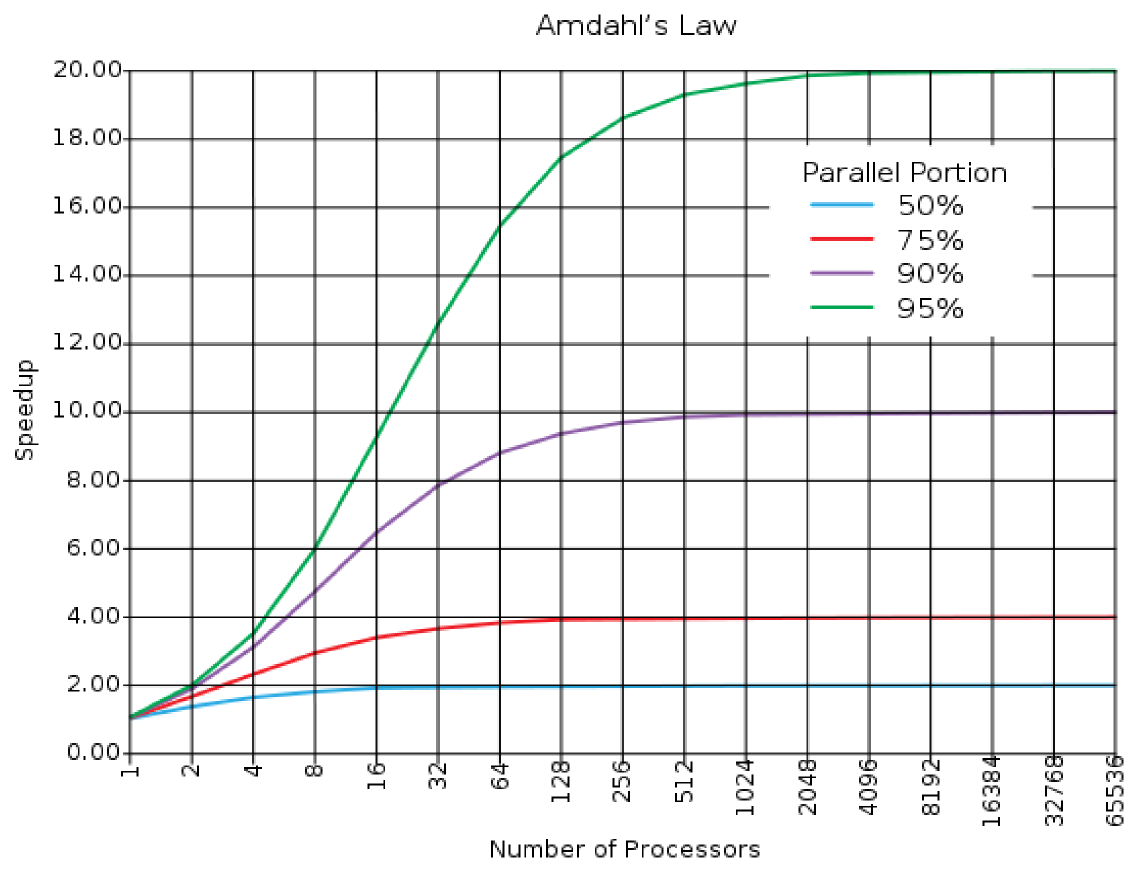


Figure 27: Amdahl's law showing relationship between maximum speedup and parallel portion with increasing number of processors for a fixed sized problem

Note: Image sourced from Wikimedia http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg and licensed for free and unrestricted use under Creative Commons license.

This behavior is presented in Figure 27, which shows a plot of the number of processors vs. speedup for a specific problem with fixed problem sizes and fixed parallel fractions. It shows that for a problem of fixed size, when the serial fraction is 50%, then the maximum speedup can only be 2, regardless of the number of processors used. Similarly for a problem of fixed sized with serial fraction as 5%, then the maximum speedup can only be 20, irrespective of the number of processors being used.

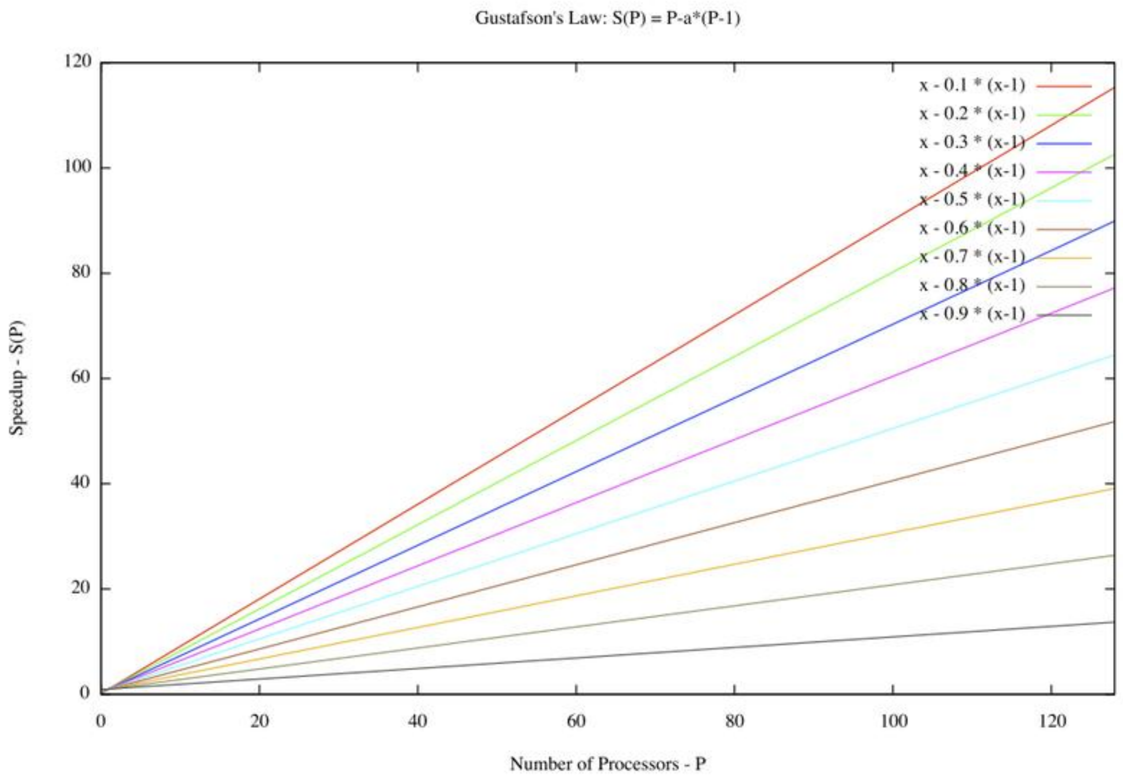


Figure 28: Gustafson's law showing scaled speedup plots

Note: Image sourced from Wikimedia http://en.wikipedia.org/wiki/File:Gustafson.png and licensed for free and unrestricted use under Creative Commons license.

The speedup of scalable problems, where the size of the problem can be scaled up, is governed by Gustafson's law [66]. The scaled speedup $S_s(p)$ is given as

$$S_s(p) = p + (1 - p)ft_s \tag{11}$$

There are two assumptions in the Gustafson's law: the parallel execution time is constant, and ft_s, which is the fixed serial fraction, is also constant and is not a function of number of processors p. Figure 28 shows a scaled speedup plot for a scalable problem with different fixed serial fractions.

In the volume-based mesh repair method presented in this research, a major portion of the solution process time is spent in obtaining a converged solution for the diffusion equations on a number of Cartesian grid columns which embed parts and pieces of input geometry. This problem becomes even more acute when holes are closely clustered together or when input geometry is of high resolution. This is due to the fact that the solution domain resolution is closely tied to the resolution of input geometry in the formulation of the diffusion equation presented in this work. One of the major objectives of this research is to study the behavior of solution processes using multi-threaded parallelization on CPUs and GPUs. The parallelization in this research has been done on an inexpensive (< $1000) PC which has a 64-bit quad-core AMD® Athlon™ II 620 processor and 4GB of RAM. The machine also has a dedicated NVIDIA® Quadro™ FX 5800 graphics card with 240 stream processor cores and 4GB of onboard GDDR3 graphics memory for GPGPU-based computations. The machine is configured with the CentOS operating systems (OS), which is a Linux-based OS, and all the data presented in this chapter was generated on the described machine. This data was generated by running the diffusion solver-based solution process on the Stanford Bunny with 10 holes in its surface [57]. The data presented in this chapter was generated on a fine grid, which uses the results of the coarser Cartesian grid for initialization of data after running the diffusion solution on the coarse Cartesian grid for a large number of iterations. It should also be noted that the diffusion solvers written for the CPU and the GPU are slightly different, as they are optimized considering very different architectures of the CPU and the GPU. In addition, it should be noted that the results are generated by embedding the whole model into one Cartesian grid, which is arranged as a number of tiles to study the performance of the diffusion solver as the problem size is scaled.

Multi-threaded Parallelization of Diffusion Equations on CPU Using OpenMP

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on a number of architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer and is supported on a number of compilers. OpenMP is natively integrated with the GNU GCC family of compilers, which is used in this research, and can be enabled using the `-fopenmp` switch at the compile time.

In this study, the entire Stanford Bunny was embedded in a single Cartesian grid of a particular size. The Cartesian grid is composed of a number of tiles in the x, y and z directions. Each tile is of equal size and contains an equal number of voxels. The tile size varies, as the Cartesian grid size is varied in this research. The total number of voxels in the Cartesian grid scales up as the Cartesian grid size increases. However, the number of tiles in the Cartesian grid has been kept fixed at 20x20x16 in the *x*, *y* and *z* directions. Furthermore, the data being presented is from a fine Cartesian grid, which interpolates data from the solution of a corresponding coarse Cartesian grid as an input to the fine Cartesian grid in a multi-grid solution-based approach. Both the tile size and the coarse Cartesian grid size was kept constant in this study. The solution run times being presented in this study are for the fine Cartesian grid only. The coarse Cartesian grid size was kept constant at 100x100x80 voxels, with 10x10x8 tiles for each Cartesian grid, irrespective of the size of the corresponding fine Cartesian grid.

Outlined in Table 3 are the data obtained from the multi-threaded parallelization of the diffusion solver on the Stanford Bunny. Column 1 in the table shows the suggested maximum Cartesian size along the maximum dimension. However, the actual size could be slightly different, as it is dependent on the ratio of maximum and minimum dimensions of a particular input model based on the bounding box sizes, chosen scaling factor, number of tiles and number of voxels per tile. Column 2 in the table shows the actual number of voxels in the Cartesian grid in which the diffusion solver is run. Columns 3 to 10 show the diffusion solver run time for 120 iterations on the fine Cartesian grid with varying numbers of threads.

Illustrated in Figure 29 is the plot of OpenMP runtime based on the data presented in Table 3. The figure shows a plot of the total number of voxels in the fine Cartesian grid vs. total run time for the diffusion solver for 120 iterations. It can be observed that the run time is linear with increasing problem size for up to 4 threads on the quad-core CPU and follows closely along with the Gustafson's law which governs maximum scaled speedup. However, as the number of threads is increased from 4, the serial fractions of the solution are no longer fixed due to thread and scheduling conflicts, and this starts affecting the run time. It can also be observed that the best run time (minimum) is achieved for 4 threads on a quad-core CPU. This happens because when the number of threads is greater than the number of available cores on the machine, the CPU has to spend a considerable amount of time in scheduling cores for competing threads solving the diffusion equation, increasing the total solution time.

Presented in Figure 30 is the plot of the OpenMP run time variation with increasing number of threads for each Cartesian grid. It can be observed from the plot that the run time for each Cartesian grid steadily declines as the number of threads is increased from 1 to 4, indicating improving speedup. However, as the number of threads increases from 4 to 5 for every Cartesian grid, the run time sharply increases, due to the change in the fixed costs for running the job. Most of the jump in the time is due to the increase in the system time spent on managing threads.

Outlined in Table 4 is the OpenMP based multi-threaded speedup for the diffusion solver for different fine Cartesian grid sizes. Shown in Figure 31 is the plot of speedup changes for variations based on changing number of threads. The best speedup, which was observed to be 2.77, was observed to be for 4 threads on Cartesian grid, with suggested size in maximum dimension as 550.

Parallelization of Diffusion Equation on GPU using CUDA

GPGPU is an exciting new area in parallel computing. It promises to provide performance similar to supercomputers from a single desktop using GPU. Computer programs customized for GPU execution emphasize high throughput and derive most of the gain in performance due to clever scheduling of many thousands of threads over hundreds of GPU processor cores. Harnessing the power of a GPU efficiently for general purpose computing requires a good understanding its architecture and requires writing special, optimized code. This also necessitates a clever partitioning of data in such a way that programming instructions on GPU can be run on any chunk of data in a random order without affecting the overall output of the computation.

One of the objectives of this work is the parallelization and optimization of the diffusion solver on a GPU, along with the evaluation and analysis of the resulting performance gain through this effort. The graphics hardware for this research has been generously donated by NVIDIA in the form of a single NVIDIA Quadro FX 5800 graphics card with 240 stream processors and 4GB of onboard GDDR3 memory. The CUDA drivers and libraries used for the GPGPU implementation are provided to developers for free by NVIDIA. Although CUDA provides a number of useful tools to write a program in standard C programming language, its use is severely restricted by a few restrictions imposed by CUDA for Quadro FX5800 generation of devices. Those restrictions are as follows:

- Limited number of ANSI C features supported by CUDA.
- Lack of support for C++ standard code.
- Restrictions on the support of higher level memory constructs, such as structures and classes.
- Absence of a true GPU-based debugger for code debugging. The current generation nvcc CUDA debugger can only be used in the device emulation mode.
- Lack of support for `std::cout` based APIs from the GPU to the user.
- Lack of support of dynamic memory allocation on the GPU from instructions executing on the GPU at runtime. This restriction severely limits what could be done on the GPU as the memory being used on the GPU has to be pre-allocated in the kernel before the code is run on the GPU.

However, despite all these restrictions, CUDA proves to be a valuable tool due to its extremely high theoretical peak performance for GPUs compared to that of even the most high-end CPUs commercially available. In this research, the diffusion solver for GPU has been optimized for GPU architecture and hence behaves slightly differently than the CPU-based diffusion solver. The diffusion solver for both the coarse and fine Cartesian grids are run on the GPU using CUDA; however, the data being presented are only for the fine Cartesian grid for a fixed number of iterations. The Cartesian grid is mapped onto a number of GPU tiles. Each of the GPU tiles is of fixed size, 16x16x*k,* number of voxels, where '*k*' is the number of voxels in the Cartesian grid along the z direction. Each of the tiles is mapped onto one stream processor on the GPU, and the number of voxels on a GPU tile corresponds to the maximum number of threads that one stream processer can spawn and handle. The thread handling on the GPU is managed by CUDA without the user having to explicitly manage them. However, CUDA requires the user to have a fine-grained knowledge of the GPU architecture to effectively use the GPU for general purpose computing to achieve the maximum throughput and highest efficiency through GPGPU implementation.

Table 5 presents the data obtained after running the GPU-optimized diffusion solver for the Stanford Bunny on fine Cartesian grid for 120 iterations. The table also presents runtime data from the diffusion solver when run on a CPU using a single thread for comparison purposes. It should, however, be noted that the diffusion solver code written for the both CPU and GPU are optimized for the architecture of their respective platforms and, as a result, there may not be exact one-to-one comparisons for each case. As a result, the run time for both the GPU and CPU was normalized for per million voxels for each suggested tile size due to the variations in the total number of voxels and, as a result, the variation in the Cartesian grid sizes, so that a comparison of run-time performances of the CPU and the GPU could be possible.

Figure 32 presents the plot of the Cartesian grid sizes for CPU and GPU vs. execution run time in seconds. It can be observed that the CPU execution time follows closely with that predicted by Gustafson’s law. However, the GPU time plot is not linear and varies when the problem size is scaled up. Possible explanations for this result include the following: 1) the hardware architecture caused maximum throughput for the GPU to correspond to the dip in the GPU time curve; 2) the internal thread scheduling and data fetching is optimized for that grid size; or 3) the mapping of the file for that particular grid size is optimal for the GPU based on its architecture.

Presented in Figure 33 is the plot of the suggested grid size along the largest dimension of the Cartesian grid vs. the solution time for the CPU and GPU. Once again, a dip in the curve could be noticed when the suggested grid size is 650 along the maximum dimension.

Shown in Figure 34 is a plot of the number of GPU tiles vs. the run time for 120 iterations for the diffusion solver on the GPU. At run time, each of the GPU tiles is mapped onto one stream processors. The GPU has 240 stream processors. When the number of tiles exceeds the number of streams processors on board the GPU, the threads mapped on to the GPU tiles wait for their turns to fetch data and perform calculations.

Presented in Figure 35 is a composite bar and line plot showing the relationship between grid size and speedup. The *y*-axis for the bar plot shows the run time per million voxels for the CPU and the GPU. The data presented has been normalized per million to obtain one-to-one correspondence between CPU and GPU for a suggested number of voxels along the largest dimension for the input model. The maximum achieved speedup was observed to be 8.86325 between CPU and GPU, while the minimum speedup was found to be 6.10873.

The results from the parallelization of the diffusion solver for CPU and GPU are presented and compared using a number of tables and plots. The CPU-based diffusion-solver code parallelization was done using the OpenMP library. The GPU-based code parallelization was done using the CUDA library and was run on a NVIDIA Quadro FX 5800 GPU. The CPU- and GPU-based diffusion solver codes were optimized for their respective architectures. As a result, the optimization required the tile size and tile arrangements on CPU and GPU to be different. The results presented in this chapter conclusively show that the GPU implementation provides superior solver performance compared to the CPU, even when compared with multi-threaded parallelization for a quad-core CPU. The maximum speedup for GPU was observed to be 8.863, while the maximum speedup after multithreading was found to be 2.771 for 4 threads on a quad core CPU.

Suggested Grid Size in Max Dimension	Total Number of voxels in millions	OpenMP diffusion solver run time for 120 iterations (in seconds)							
		1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads
150	3.28	60.96	46.78	42.52	37.50	84.99	87.39	99.03	96.40
200	6.40	109.42	76.22	70.05	62.01	106.55	108.37	115.31	132.39
250	14.06	223.14	151.18	126.60	115.23	163.92	173.21	157.17	164.39
300	21.60	326.05	212.43	165.56	147.63	208.93	217.03	224.50	194.70
350	35.25	511.18	313.12	244.73	220.58	290.48	279.54	284.18	294.35
400	51.20	718.11	428.31	338.39	287.68	391.56	398.44	353.30	411.93
450	74.48	1030.58	623.36	474.95	407.59	536.20	506.48	512.17	455.08
500	100.00	1365.36	835.74	631.30	538.84	704.42	712.20	626.83	637.19
550	135.48	1823.95	1050.35	827.91	658.15	850.65	857.34	754.50	760.45
600	172.80	2282.76	1293.11	966.46	832.86	1118.14	993.97	998.11	873.10

Table 3: Data obtained from OpenMP based multi-threaded parallelization for diffusion equation on Stanford Bunny

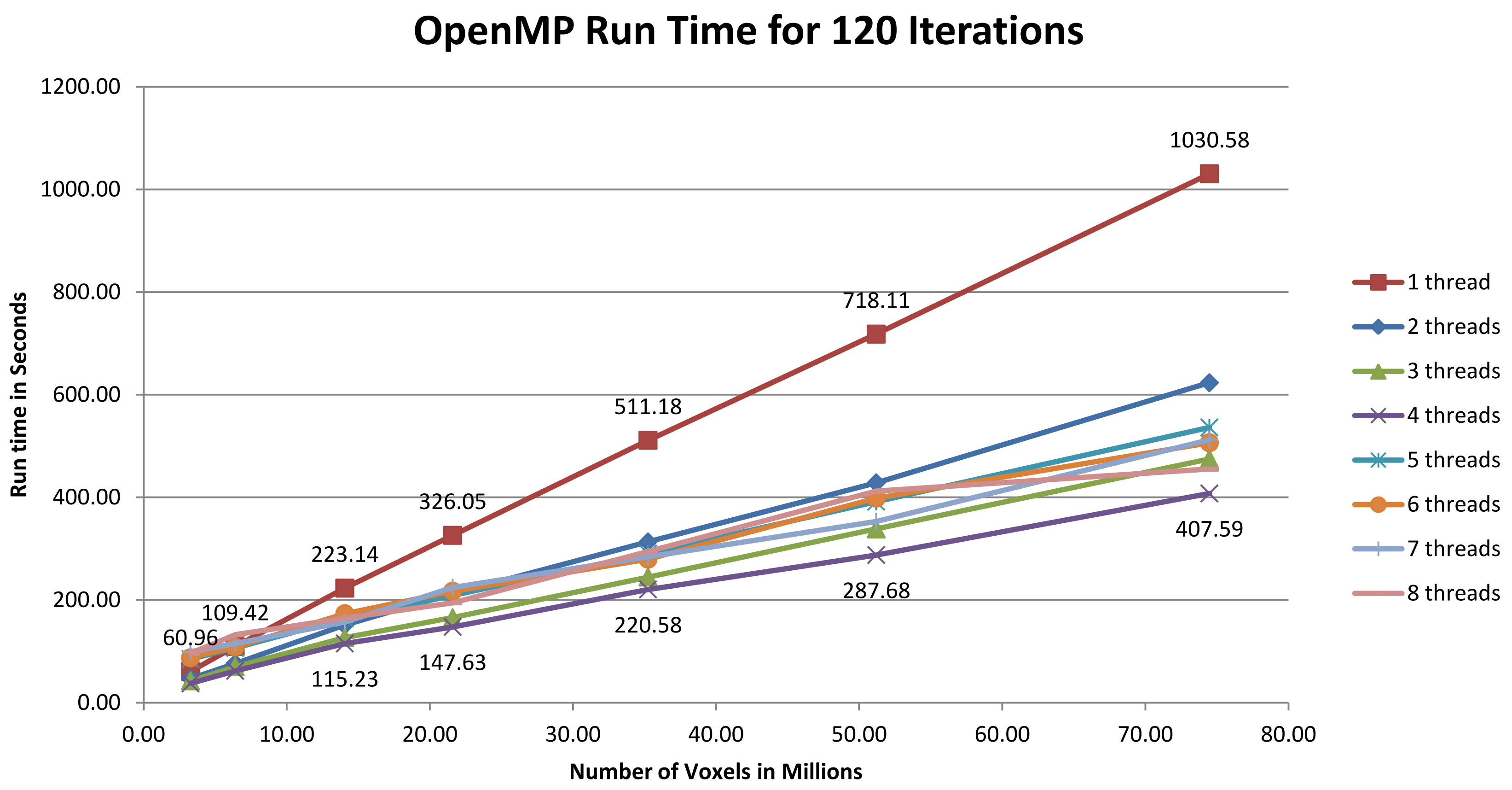


Figure 29: OpenMP-based multi-threaded implementation for 120 iterations on Stanford Bunny showing plot of number of voxels in millions vs. run time in seconds

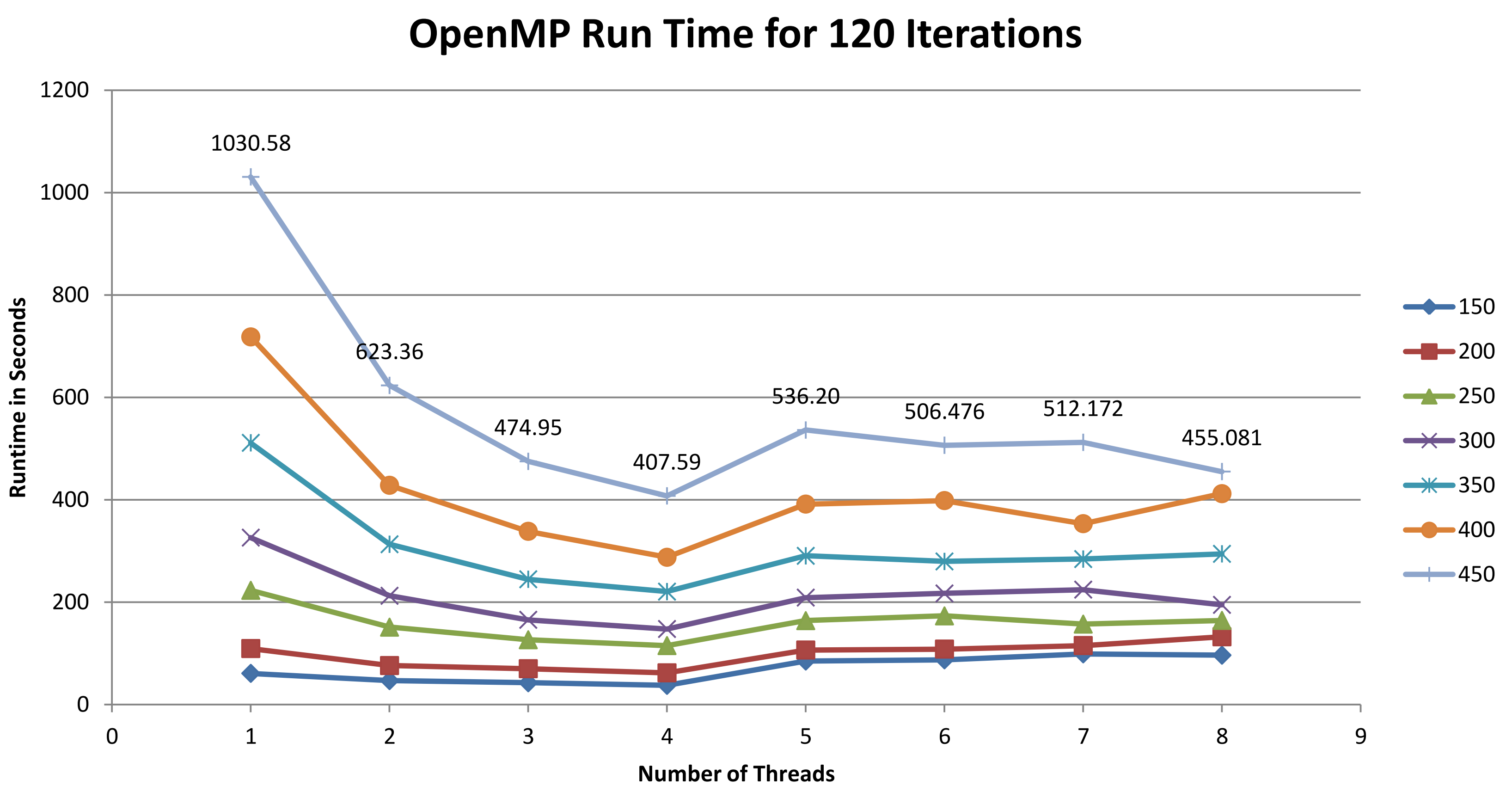


Figure 30: OpenMP-based multi-threaded implementation for 120 iterations on Stanford Bunny showing plot of number of threads vs. run time in seconds

Suggested Tile Size in Max Dimension	OpenMP Multi-Threading Speedup							
	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads
150	1.0	1.303	1.434	1.626	0.717	0.698	0.616	0.632
200	1.0	1.436	1.562	1.764	1.027	1.010	0.949	0.826
250	1.0	1.476	1.763	1.936	1.361	1.288	1.420	1.357
300	1.0	1.535	1.969	2.209	1.561	1.502	1.452	1.675
350	1.0	1.633	2.089	2.317	1.760	1.829	1.799	1.737
400	1.0	1.677	2.122	2.496	1.834	1.802	2.033	1.743
450	1.0	1.653	2.170	2.529	1.922	2.035	2.012	2.265
500	1.0	1.634	2.163	2.534	1.938	1.917	2.178	2.143
550	1.0	1.737	2.203	2.771	2.144	2.127	2.417	2.399
600	1.0	1.765	2.362	2.741	2.042	2.297	2.287	2.615

Table 4: Speedup of OpenMP-based multi-threaded diffusion solver for 120 iterations on Stanford Bunny

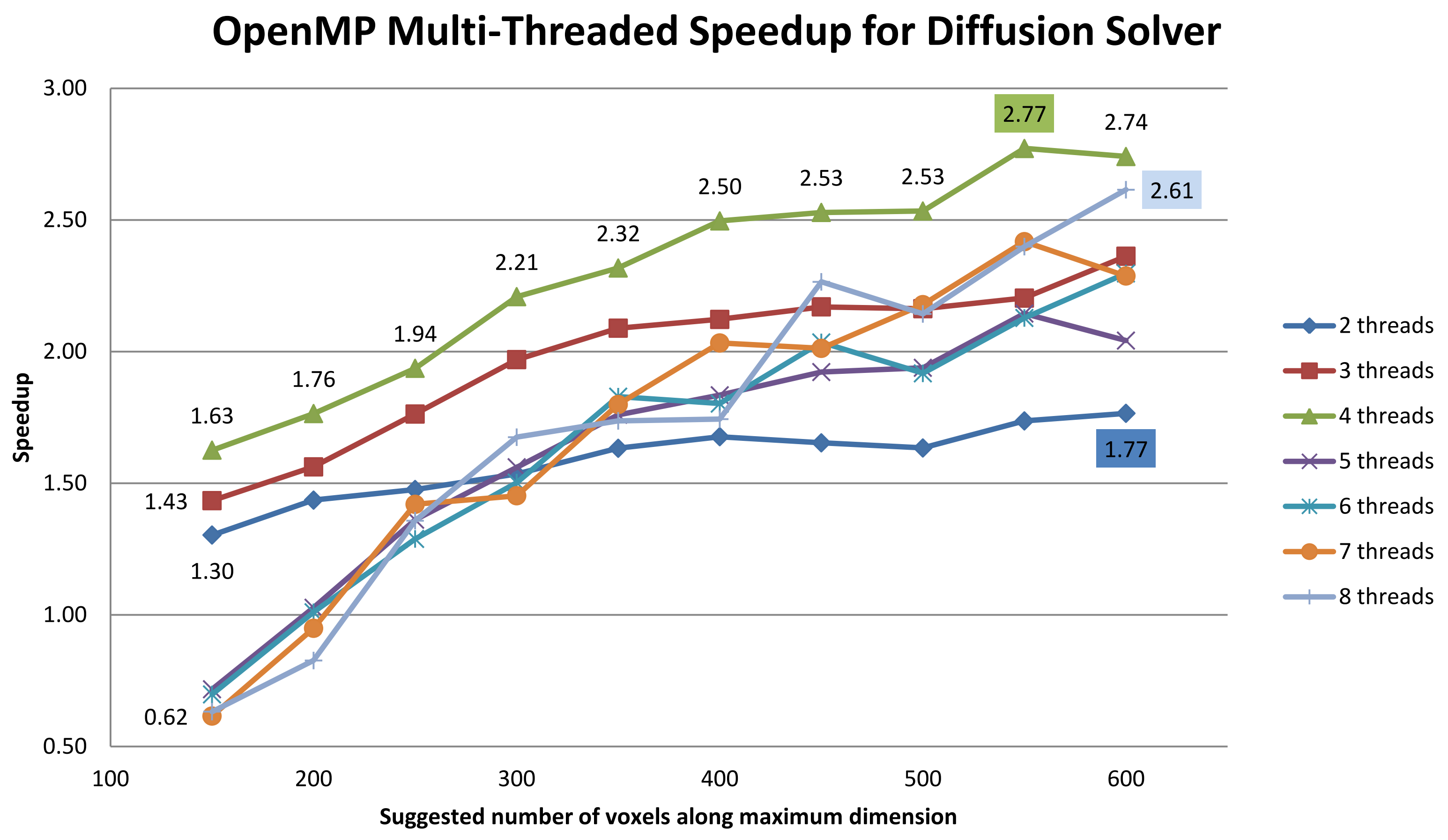


Figure 31: Speedup plot of OpenMP-based multi-threaded diffusion solver for 120 iterations on Stanford Bunny

Suggested Tile Size in Max Dimension	Number of Tiles mapped on GPU Stream Processors	Number of GPU Voxels in Millions	Run Time in Seconds for 120 iterations			Number of CPU Voxels in Millions	Run Time in Seconds for 120 iterations		GPU Speedup
			GPU unoptimized time	GPU optimized time	GPU optimized time normalized		CPU Execution time in Seconds	CPU Normalized time per million voxel	
150	100	3.278	6.353	5.23263	1.5962874	3.28	44.640	13.622955	8.53415
200	169	6.922	13.534	11.1449	1.6100135	6.40	80.912	12.642484	7.85241
240	225	11.059	22.006	18.0063	1.6281738	--	--	--	--
250	256	13.632	27.340	22.4419	1.6463265	14.06	165.575	11.775646	7.15268
300	361	22.180	45.082	37.0587	1.6708311	21.60	249.180	11.536111	6.90441
336	441	30.708	62.576	51.439	1.6751173	--	--	--	--
350	484	33.702	68.677	56.5292	1.6773298	35.25	394.922	11.20308	6.67912
400	625	51.200	106.441	87.2626	1.7043477	51.20	557.800	10.894531	6.39220
416	676	58.147	120.891	99.1733	1.7055676	--	--	--	--
450	812	73.171	150.713	123.765	1.6914511	74.48	808.777	10.858516	6.41965
480	900	88.474	184.000	151.517	1.7125674	--	--	--	--
500	992	101.581	212.366	173.637	1.7093453	100.00	1073.430	10.7343	6.27977
544	1156	127.844	265.682	217.986	1.7050937	--	--	--	--
550	1225	135.475	280.674	230.182	1.6990736	135.48	1406.120	10.379184	6.10873
592	1369	162.615	307.822	254.154	1.5629185	--	--	--	--
600	1444	177.439	304.680	252.188	1.4212659	172.80	1765.240	10.215509	7.18761
640	1600	203.162	307.350	257.141	1.2656944	--	--	--	--
650	1681	220.332	298.888	253.05	1.1484941	223.03	2270.280	10.179395	8.86325
700	1936	269.615	464.038	385.116	1.4283923	266.56	2702.080	10.136855	7.09669

Table 5: Execution time for diffusion solver on CPU and GPU for Stanford Bunny

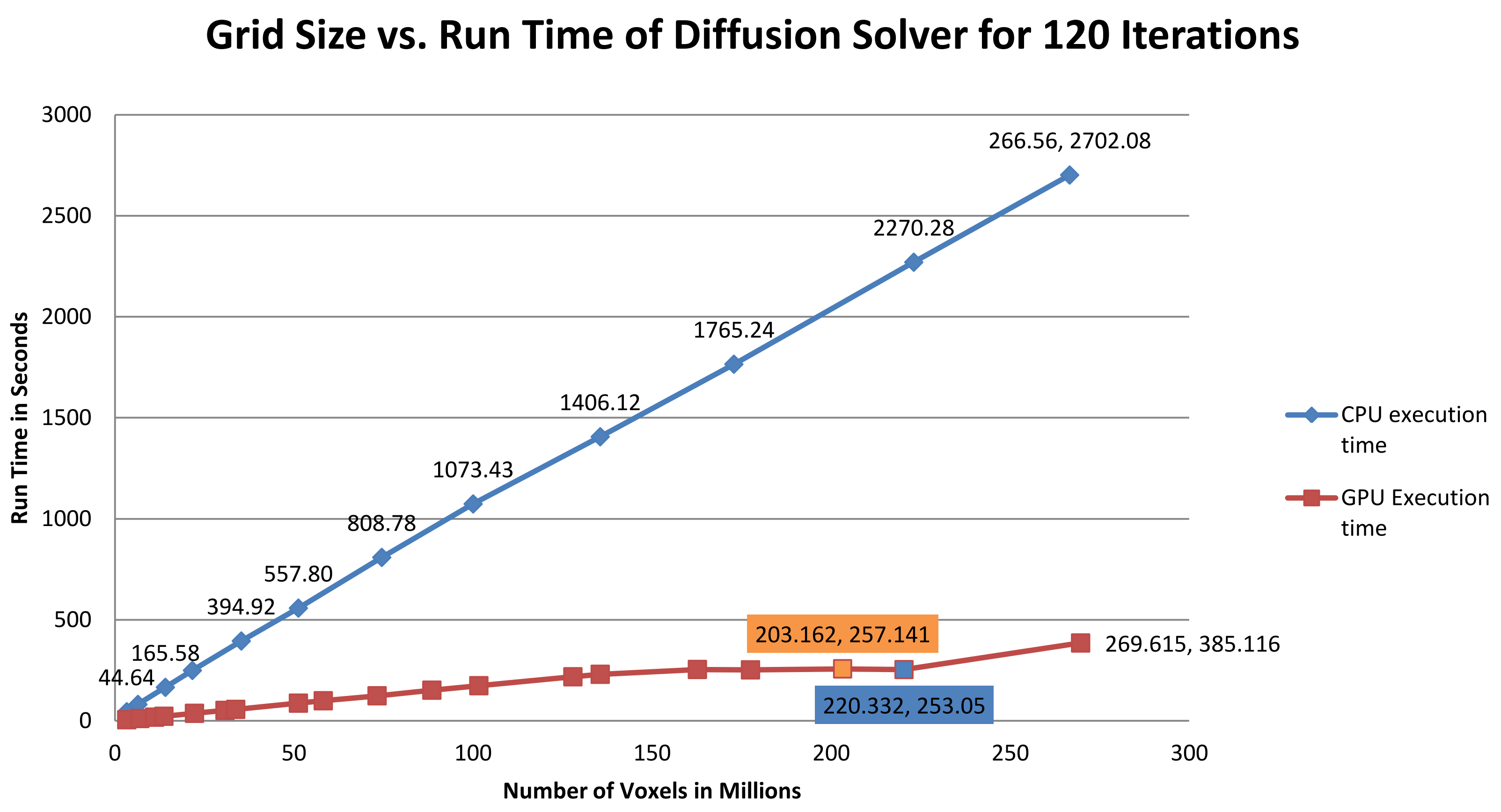


Figure 32: Plot of grid size in million voxels vs. run time for Diffusion solver for 120 iterations on GPU and CPU.

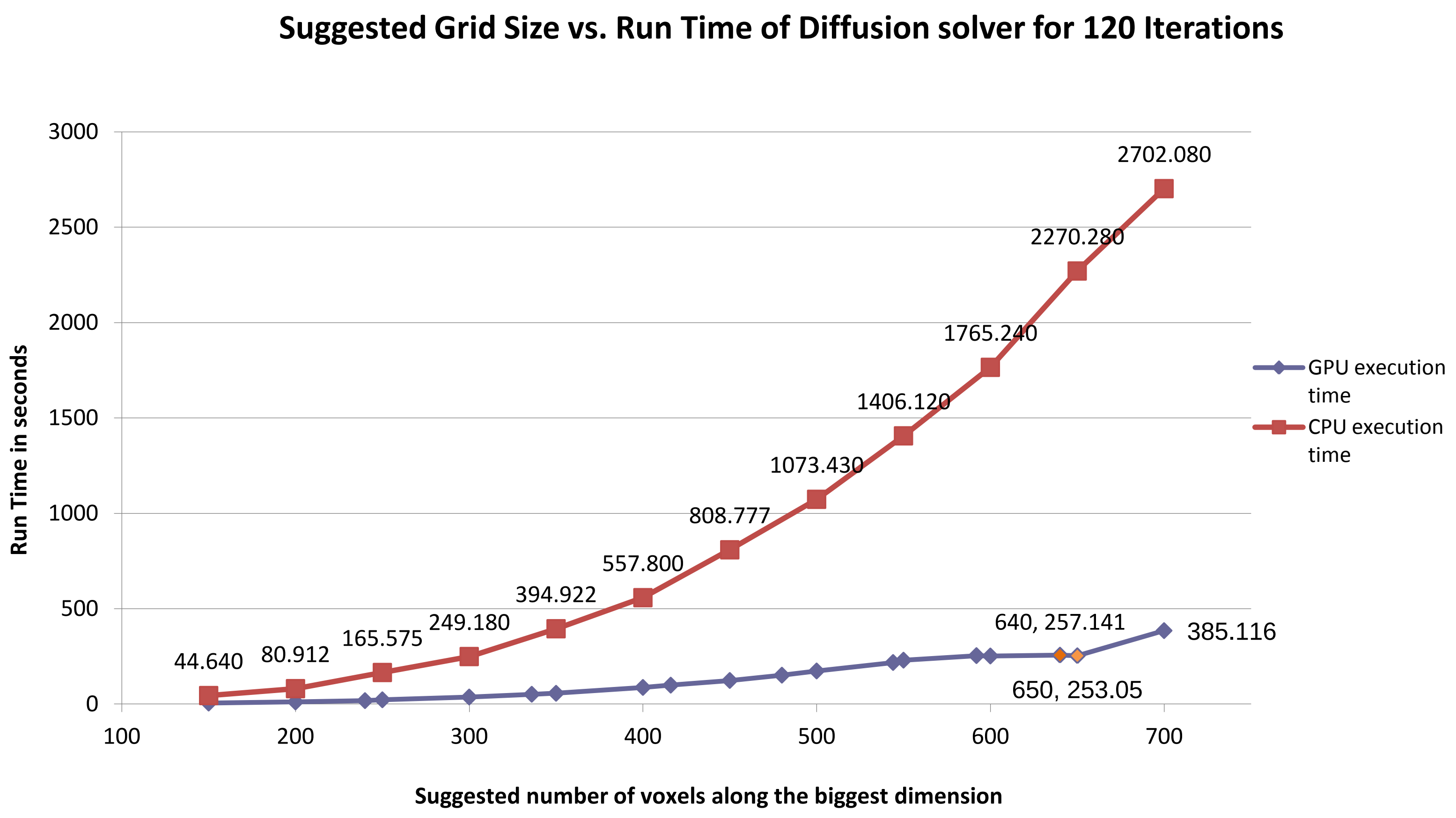


Figure 33: Plot of grid size along the biggest dimension vs. run time for Diffusion solver for 120 iterations on GPU and CPU.

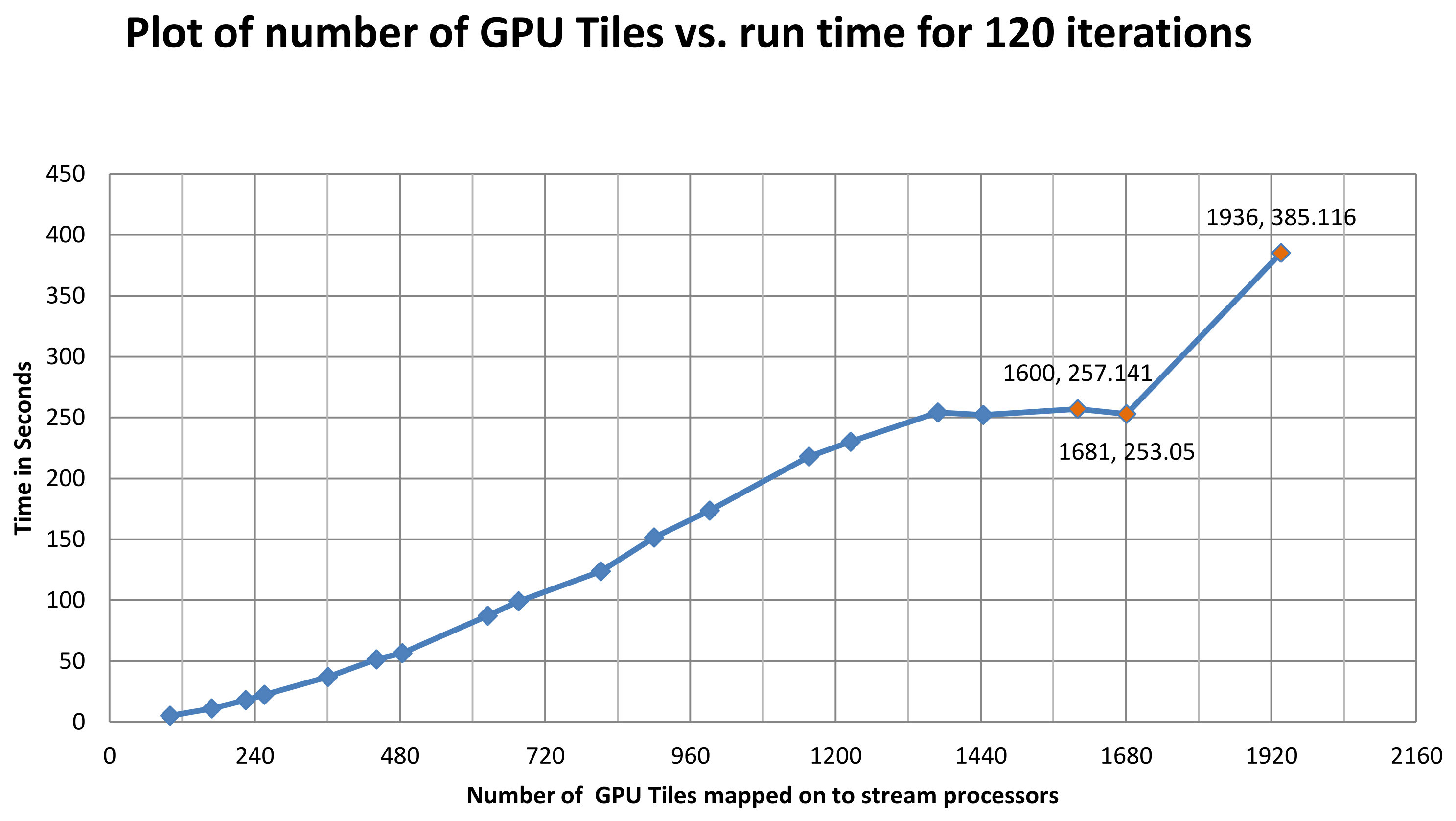


Figure 34: Plot of number of solution tiles on GPU with each tile mapped on a single stream processor vs. run time for Diffusion solver for 120 iterations.

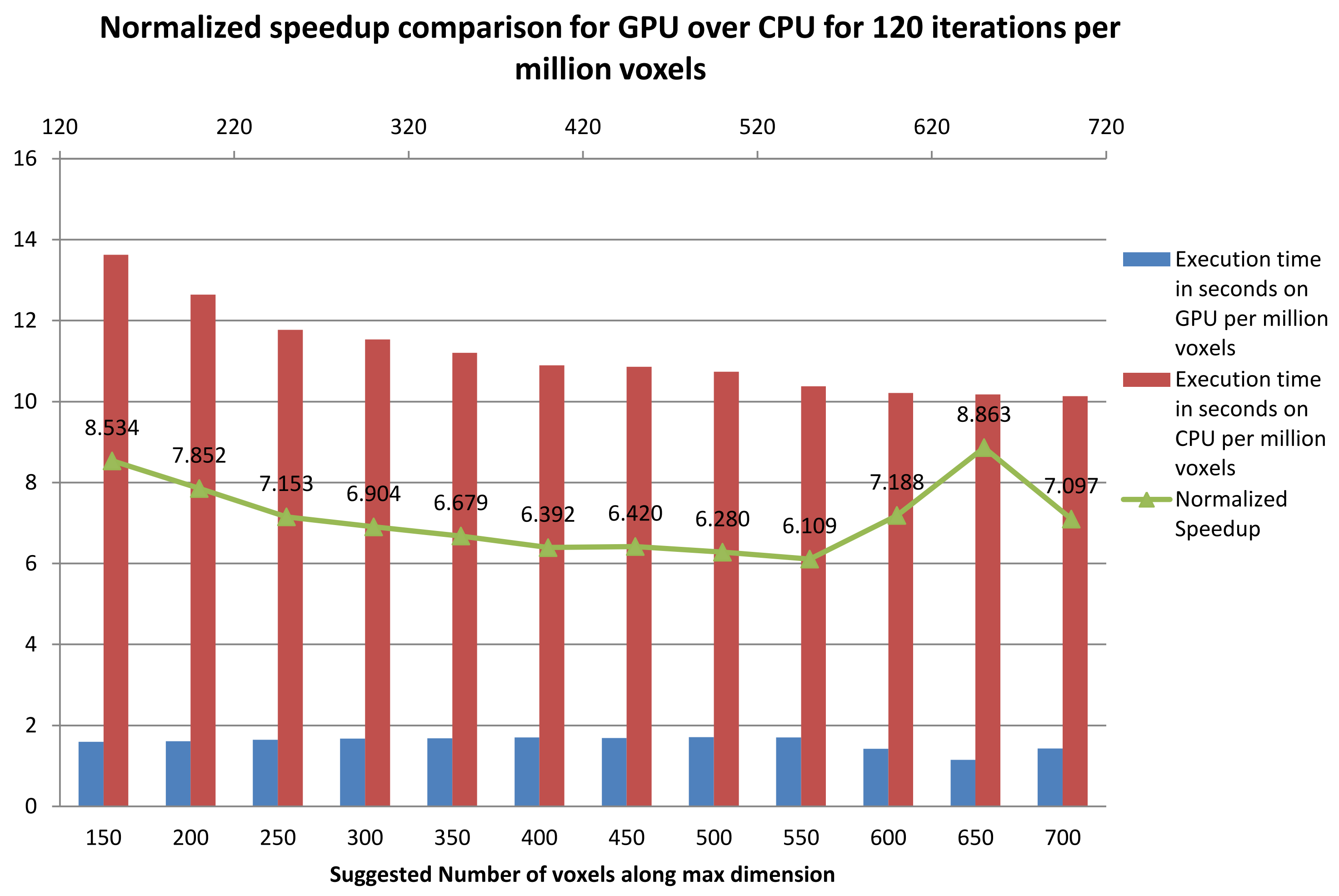


Figure 35: Plot of normalized speedup comparison for CPU and GPU for 120 iterations per million voxels on the solution grid vs. grid size along the maximum dimension.

CHAPTER 7

SUMMARY

Geometry deficiencies are major roadblocks in today's computational engineering simulation cycles. They can require tedious and laborious efforts to repair the deficiencies. This study presents completely automatic volume-based and hybrid algorithms for geometric hole patching, which could be used to repair meshes with topologically complex holes. These algorithms are able to repair holes on geometrically and topologically complex models which otherwise cannot be completely and satisfactorily repaired by either surface-based or volume-based methods in isolation. The solution of the diffusion equation is an important part of both the volume-based as well as hybrid approach to hole filling and consumes a large amount of computer time. The solution process is repetitive and, as such, is ideal for parallelization. The diffusion solver was parallelized, and its performance was analyzed on both the CPU and the GPU and presented here. The dissertation is organized as follows:

Chapter 1 presents a brief introduction along with the motives for this research. Chapter 2 starts with a discussion of the available hole-patching and mesh-repair-related research published in a number of research papers and journals. Some of the available surface reconstruction algorithms are also discussed in Chapter 2. Chapter 3 presents the research and conclusions as described in a number of papers describing the evolution and advancements in the area of general purpose computing applications. The chapter also describes the requirements for using GPU hardware for general purpose computing. A brief description of the CUDA (Compute Unified Device Architecture) library was presented, which was later used for parallelization of diffusion solver on GPU as presented in Chapter 6. In Chapter 4, a new volume-based method was presented that uses a solution of finite volume formulation of diffusion equation, as described in Appendix A, followed by contouring to generate surfaces in the region surrounding the holes. Patches and point sets are extracted from the generated surface and are used as an input for a Poisson surface reconstruction algorithm to generate watertight, reconstructed discrete surface mesh. Chapter 5 discusses a hybrid approach to geometric hole filling that uses a surface-based approach, which is an improvement over the previous published work [12], [13] and [14] and the volume-based approach described in chapter 4. The hybrid approach combines the best features of both surface- and volume-based algorithms for mesh repair. The usefulness of the volume-based algorithm as described in chapter 4 and hybrid approach as described in chapter 5 was demonstrated using the Stanford Bunny, Laurent Hand and Chinese Lion models. Results presented in the chapters 4 and 5 have been accepted for publication in the proceedings of the 20th International Meshing Roundtable [24]. Chapter 6 discusses multi-core, multi-threaded parallelization of the diffusion equation solver on the CPU and the GPU. The results for the parallelization were generated on a single computational geometry, Stanford Bunny [57]. The results were studied and compared between the GPU and CPU, and their performance and relative speedups are presented in chapter 6. The results show that the

GPU offers a superior performance advantage over the CPU in iterative and repetitive SPMD-type calculations.

With the completion of the research and the achievements needed to meet the objectives set in the dissertation proposal, the unique contributions of this research can be identified as the development of a volume-based hole-patching algorithm using the diffusion equation, as well its inclusion in a hybrid approach to mesh repair that combines the best features of both surface- and volume-based algorithms. The approach demonstrated here is similar in approach to that described by Curless and Levoy [18] and Davis et al. [19]. Those similar volume-based approaches require extra information in the form of line-of-sight information from ranging devices for setting boundary conditions and geometry repair, which is not required in the research presented here.

The unique contributions of this research are due to the following:

- Hole Patching for Geometry: Geometry deficiencies are major roadblocks in today's computational engineering simulation cycles. They can require tedious and laborious efforts to repair the deficiencies. This research effort contributes to the state of the art by facilitating the repair process specific to the hole-patching issue using a volumetric approach without requiring any extraneous information, such as ranging data as used in Curless and Levoy [18] for *carving in space and time* in mesh repair. In addition, this research presents a hybrid approach that uses both surface-based and volume-based approaches to repair complex geometries with topologically complex holes. The usefulness of the volume-based method and hybrid approach is demonstrated with a number of examples.

- Hole Identification and Isolation Algorithms: It is challenging to identify and isolate a hole region on a geometric model, especially when the hole is topologically and geometrically complex. This research develops and presents a Cartesian grid-based algorithm to identify and extract the geometrical information for the region. It builds on the method for identification of holes as presented in Kumar et al. [12] and extends it to include topologically complex holes by bringing in the notion of the connectivity of input fragments, thereby identifying simple holes and isolating fragments from the identified simple holes.
- Localized "*Column Grid*" Approach: The approach developed here isolates each hole region and builds a column grid around it. The hole-patching algorithm is applied to this column grid independently from others, making the problem suitable for parallel computation.
- Parallelization of diffusion solvers on CPU and GPU: Since the diffusion equation could be solved on each voxel repeatedly until the solution domain has reached a desired level of convergence, this type of process can be easily parallelized on both the CPU and the GPU. CPU-based parallelization and performance study has been done by multi-threading using the OpenMP library. GPU-based parallelization was done using NVIDIA's CUDA library. GPU-based parallel computation for this type of application has not been done before to the best of our knowledge. Therefore, the research and implementation of the algorithms on the GPUs is another unique contribution of this research.

Scope for Future Work

This work presented here is useful for repairing meshes with holes and small intersections on dirty geometries with the presence of isles. However, this method will not be able to completely repair the input mesh and may provide unexpected results when mesh overlaps are significant and of large scale. As a result, there is a need for further development of the solution process to resolve large-scale overlaps prior to mesh repair.

This dissertation presents the parallelization of the diffusion solver for both the CPU and the GPU. However, the parallelized solvers are used as stand-alone modules for performance comparisons. These parallelized solvers could be integrated with the main code to leverage higher performance from the solution process.

The diffusion solver has been optimized and parallelized for a single NVIDIA GPU device. The performance study of the solvers shows superior performance of the GPU-based solver. However, a much larger performance gain could possibly have been achieved by better load balancing and leveraging the cache hierarchy on the GPU device. The scope for the parallelization effort in this research work has been limited, and there is potential to increase the performance envelope of the GPU-based solver to achieve higher performance gain. Additional work may be done to parallelize the GPU solver code on multiple GPU devices.

LIST OF REFERENCES

[1] Botsch M, Pauly M, Kobbelt L, Alliez P, Lévy B, Bischoff S and Rössl C (2007) Geometric Modeling Based on Polygonal Meshes. ACM SIGGRAPH 2007 Courses - International Conference on Computer Graphics and Interactive Techniques

[2] Turk G and Levoy M (1994) Zippered Polygon Meshes from Range Images. In Proceedings of ACM SIGGRAPH 94, 311-318

[3] Barequet G and Sharir M (1995) Filling Gaps in the Boundary of a Polyhedron. Computer Aided Geometric Design, 12: 207-229

[4] Barequet G and Kumar S (1997) Repairing CAD Models,. Proceedings of IEEE Visualization 97, 363-370

[5] Guéziec A, Taubin G, Lazarus F and Horn B (2001) Cutting and Stitching: Converting Computer sets of Polygons to Manifold Surfaces. IEEE Transactions on Visualization and Graphics, 7(2):136–151

[6] Guskov I and Wood JZ (2001) Topological Noise Removal. In Proceedings of Graphics Interface 2001, 19-26

[7] Borodin P, Novotni M and Klein R (2002) Progressive Gap Closing for mesh repairing. In J. Vince and R. Earnshaw, editors, Advances in Modelling, Animation and Rendering, Springer Verlag, 201-213

[8] Liepa P (2003) Filling Holes in Meshes. Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry processing, Eurographics Association, 200-205

[9] Jun Y (2005) A Piecewise Hole Filling Algorithm in Reverse Engineering. Computer Aided Design, 37:263-270

[10] Branch J, Prieto F and Boulanger P (2006) A Hole-Filling Algorithm for Triangular Meshes using Local Radial Basis Function. Proceedings of the 15th International Meshing Roundtable, Springer, 411-431

[11] Pernot JP, Moraru G and Vernon P (2006) Filling holes in meshes using a mechanical model to simulate the curvature variation minimization. Computer and Graphics, 30(6):892-902

[12] Kumar A, Shih AM, Ito Y, Ross DH and Soni BK (2007) A Hole-Filling Algorithm Using Non-Uniform Rational B-Splines. Proceedings of 16th International Meshing Roundtable, Springer Berlin Heidelberg, ISBN: 978-3-540-75102-1, 169-182

[13] Kumar A and Shih AM (2009) Patching Topologically Simple Holes in Unstructured Mesh Using Non-Uniform Rational B-Splines. ASME Early Career Technical Journal, 8(1): 21.1-21.8

[14] Kumar A, Ito Y, Yu T, Ross D and Shih AM (2011) A Novel Hole Patching Algorithm for Discrete Geometry using Non-Uniform Rational B-Spline. International Journal for Numerical Methods in Engineering. Published Online

[15] Akenine-Möller, T (2005) Fast 3D Triangle-Box Overlap Testing. In ACM SIGGRAPH 2005 Courses (Los Angeles, California, July 31 - August 04, 2005). J. Fujii, Ed. SIGGRAPH '05. ACM, New York, NY

[16] Akenine-Möller (2011) AABB-triangle overlap test code. Source code is located at `http://jgt.akpeters.com/papers/AkenineMoller01/tribox.html`

[17] Gonzalez CR Woods ER Digital Image Processing. Prentice Hall; 3rd edition, ISBN-13**:** 978-0131687288

[18] Curless B and Levoy M (1996) A Volumetric Method for Building Complex Models from Range Images. Computer Graphics, 30:303-312

[19] Davis J, Marschner SR, Garr M and Levoy M (2002) Filling Holes in Complex Surfaces using Volumetric Diffusion. Proceedings of First International Symposium on 3D Data Processing, Visualization, Transmission, 2002, 428-861

[20] Nooruddin FS and Turk G (2003) Simplification and repair of polygonal models using volumetric techniques. IEEE Transactions on Visualization and Computer Graphics, 9(2):191-205

[21] Ju T (2004) Robust Repair of Polygonal Models. Proceedings of ACM SIGGRAPH, 2004, ACM Transactions on Graphics, 23:888-895

[22] Bischoff S, Pavic D and Kobbelt L (2005) Automatic restoration of polygon models. Transactions on Graphics, 24(4):1332-1352

[23] Podolak J and Rusinkiewicz S (2005) Atomic volumes for mesh completion. In Symposiuon on Geometry Processing

[24] Kumar A and Shih AM (2011) Hybrid Approach for Repair of Geometry With Complex Topology. Accepted for the proceedings of 20th International Meshing Round Table, October 2011

[25] Kobbelt LP, Vorsatz J, Ulf L and Seidel HP (1999) A Shrink Wrapping Approach to Remeshing Polygonal Surfaces. Computer Graphics Forum (Eurographics '99), 18:119-130.

[26] Klincsek G (1980) Minimal Triangulation of Polygonal Domain. Annals of Discrete Mathematics, 9:121-123

[27] Murali MT and Funkhouser AT (1997) Consistent solid and boundary representations from arbitrary polygonal data. In Proc. Symposium on Interactive 3D Graphics, 155-162

[28] Amenta N and Bern M (1999) Surface Reconstruction by Vornoi Filtering. Discrete & Computational Geometry, 22(4):481-504

[29] Amenta N, Choi SC and Kolluri R (2001) The powercrust, unions of balls, and the medial axis transform. Computational Geometry: Theory and Applications, 19:127-153

[30] Carr CJ, Beatson KR, Cherrie BJ, Mitchell JT, Evans RT, Fright RW and McCallum CB (2001) Reconstruction and representation of 3D objects with Radial Basis functions, In Proceedings of ACM SIGGRAPH '01, 67-76

[31] Bruno L (2003) Dual Domain Extrapolation. ACM Transactions on Graphics (SIGGRAPH), 22:364-369

[32] Dey KT and Goswami S (2003) Tight Cocone: A Water-tight Surface Reconstructor. Journal of Computing and Information Science in Engineering, 3(4):302-307

[33] Dey KT and Goswami S (2004) Provable surface reconstruction from noisy samples. Computational Geometry, 35(12):124-141

[34] Shen C, O'Brien FJ and Shewchuk RJ (2004) Interpolating and approximating implicit surfaces from polygon soup. In Proceedings of ACM SIGGRAPH'04, 896-904

[35] Casciola G, Lazzaro D, Montefusco BL and Morigi S (2005) Fast surface reconstruction and hole filling using positive definite radial basis functions. Journal of Numerical Algorithms, 39:289-305

[36] Kazhdan M, Bolitho M and Hoppe H (2006) Poisson Surface Reconstruction. In Proceedings of the fourth Eurographics symposium on Geometry processing (SGP '06), Switzerland, 61-70

[37] Mullen P, Goes DF, Desbrun M, Cohen-Steiner D and Alliez P (2010) Signing the Unsigned: Robust Surface Reconstruction from Raw Point Sets. Eurographics Symposium on Geometry Processing 2010, 29(5):1733-1741

[38] Doria D and Gelas A (2010) Poisson Surface Reconstruction for VTK. The VTKJournal. http://www.insight-journal.org/download/viewpdf/718/2/download

[39] Kobbelt L, Botsch M, Schwanecke U and Seidel HP (2001) Feature sensitive surface extraction from volume data. In Proc. of ACM SIGGRAPH 01, 57-66

[40] Lorenson EW and Cline EH (1987) Marching Cubes: A High Resolution 3D Surface Construction Algorithm. ACM Computer Graphics, 21(3): 163-169

[41] Halfhill RT (2008) Parallel processing with CUDA: NVIDIA’s high-performance computing platform uses massive multithreading. Microprocessor Report, January 28 2008, http://www.NVIDIA.com/docs/IO/55972/220401_Reprint.pdf

[42] Luebke D (2008) CUDA: Scalable parallel programming for high-performance scientific computing. 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, 2008, ISBN 978-1-4244-2002-5, 836-838

[43] Owens DJ, Leubke D, Govindraju N, Harris M, Kruger J, Lefohn EA and Purcell JT (2007) A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 26(1): 80-113

[44] Owens DJ, Houston M, Luebke D, Green S, Stone EJ and Phillips CJ (2008) GPU Computing. Proceedings of the IEEE In Proceedings of the IEEE, 96(5): 879-899

[45] Ryoo S, Rodrigues IC, Stone SS, Stratton AJ, Ueng S, Baghsorkhi SS and Hwu WW (2008) Program optimization carving for GPU computing. Journal of Parallel and Distributed Computing, 68(10): 1389-1401

[46] Wang L, Huang Y, Chen X and Zhang C (2008) Task Scheduling of Parallel Processing in CPU-GPU Collaborative Environment. In Proceedings of the 2008 international Conference on Computer Science and information Technology, ICCSIT, IEEE Computer Society, Washington, DC

[47] Garland M, L2 GS, Nickolls J, Anderson J, Hardwick J, Morton S, Phillips E, Zhang Y and Volkov V (2008) Parallel Computing Experiences with CUDA. IEEE Micro, 28(4):13-27

[48] Messmer P, Mullowney PJ and Granger BE (2008) GPULib: GPU Computing in High-Level Languages. Computing in Science & Engineering, 10(5):70-73

[49] Pharr M and Fernando R (2005) GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, ISBN-10: 0321335597, ISBN-13: 978-0321335593

[50] Dongarra J, Peterson G, Tomov S, Allerd J, Natoli V and Richie D (2008) Exploring New Architectures in Accelerating CFD for Air Force Applications. DoD HPCMP User Group Conference 2008

[51] (2010) NVIDIA CUDA Compute unified Device Architecture Programming guide Version 2.0. http://developer.download.NVIDIA.com/ compute/cuda/ 2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf

[52] (2011) NVIDIA® CUDA™ parallel computing architecture. http://www.NVIDIA.co m/object/GPU_Computing.html

[53] Glaskowsky NP (2011) NVIDIA's Fermi: The First Complete GPU Computing Architecture. http://www.NVIDIA.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA%27s_Fermi-The_First_Complete_GPU_Architecture.pdf

[54] Taubin G, Zhang T and Golub G (1996) Optimal Surface Smoothing As Filter Design, In proceedings, Fifth International Conference on Computer Vision, 283-292

[55] M Botsch (2011) Extended Marching Cubes implementation. http://www.graphics.rwth-aachen.de/index.php?id=17

[56] (2011) Diffusion Equation on Wikipedia. http://en.wikipedia.org/wiki/Diffusion_equation

[57] (2011) Stanford Bunny at The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep/

[58] Saboret L, Attene M and Alliez P (2011) Laurent Hand at AIM@SHAPE Shape Repository. http://shapes.aim-at-shape.net/viewgroup.php?id=785

[59] Saboret L, Attene M and Alliez P (2011) Chinese Lion at AIM@SHAPE Shape Repository. http://shapes.aim-at-shape.net/viewgroup.php?id=783

[60] (2011) What is VTK? http://vtk.org/what-is-vtk.php

[61] (2011) vtkImageMarchingCubes Filter http://www.vtk.org/doc/nightly/html/classvtkImageMarchingCubes.html

[62] (2011) vtkWindowedSincPolyDataFilter http://www.vtk.org/doc/nightly/html/classvtkWindowedSincPolyDataFilter.html

[63] (2011) Aim@Shape repository. http://shapes.aim-at-shape.net

[64] (2011) Paraview, An open-source, multi-platform data analysis and visualization application. http://www.paraview.org/

[65] (2011) Amdahl’s Law. http://en.wikipedia.org/wiki/Amdahl's_law

[66] (2011) Gustafsons’s Law. http://en.wikipedia.org/wiki/Gustafson's_law

[67] (2011) The Digital Michelangelo Project http://graphics.stanford.edu/projects/mich/

[68] (2011) The OpenMP Library http://openmp.org/wp/

[69] Thompson FJ, Soni KB and Weatherill PN (1998) Handbook of Grid Generation. ISBN 0-8493-2687-7, CRC Press.

APPENDIX A

DIFFUSION EQUATION

Diffusion is a time-dependent process, constituted by random motion of given entities and causing the statistical distribution of these entities to spread in space. The concept of diffusion is tied to the notion of mass transfer, driven by a concentration gradient. The diffusion equation can be obtained easily from this when combined with Fick's first law, which assumes that the flux of the diffusing material in any part of the system is proportional to the local density gradient. The diffusion equation is a partial differential equation that describes density fluctuations in a material undergoing diffusion. The diffusion equation is given as

$$\frac{\partial \emptyset}{\partial t} = \alpha \nabla^2 \emptyset + S \tag{12}$$

Where α is a constant and S is a source term. In this formulation, it is assumed that there are no source terms, Hence the equation becomes

$$\frac{\partial \emptyset}{\partial t} = \alpha \nabla^2 \emptyset \tag{13}$$

The finite volume formulation of the diffusion equation over a volume **Ω** could be written as

$$\int \frac{\partial \emptyset}{\partial t} d\Omega = \alpha \int \nabla^2 \emptyset \, d\Omega \tag{14}$$

Applying Green's theorem the equation changes over a closed area A in the form of,

$$\int \frac{\partial \emptyset}{\partial t} d\Omega = \alpha \oint \vec{\nabla}\emptyset . \hat{n} \, dA \tag{15}$$

The above equation can be approximated as following using forward difference in time, to obtain an explicit formulation in the form of

$$=> \frac{\emptyset_{i,j,k}^{n+1} - \emptyset_{i,j,k}^{n}}{\Delta t} \Omega_{\mathrm{i,j,k}} = \alpha \sum_{i=1}^{n} \sum_{\mathrm{m}} \vec{\nabla}\emptyset . \hat{\mathrm{n}} \; A_{i,m} \tag{16}$$

where *i* is the number of direction and m is the index of area along *i* direction.

For a 3D problem, the equation after discretization will approximate to.

$$\begin{aligned} => \frac{\emptyset_{i,j,k}^{n+1} - \emptyset_{i,j,k}^{n}}{\Delta t} \Omega_{\mathrm{i,j,k}} &= \\ \alpha \left(\sum \frac{\emptyset_{i+1,j,k}^{n} - \emptyset_{i,j,k}^{n}}{\Delta x} min(A_i, A_{i+1}) - \sum \frac{\emptyset_{i,j,k}^{n} - \emptyset_{i-1,j,k}^{n}}{\Delta x} min(A_{i-1}, A_i) \right) & \\ + \alpha \left(\sum \frac{\emptyset_{i,j+1,k}^{n} - \emptyset_{i,j,k}^{n}}{\Delta y} min(A_j, A_{j+1}) - \sum \frac{\emptyset_{i,j,k}^{n} - \emptyset_{i,j-1,k}^{n}}{\Delta y} min(A_{j-1}, A_j) \right) & \\ + \alpha \left(\sum \frac{\emptyset_{i,j,k+1}^{n} - \emptyset_{i,j,k}^{n}}{\Delta z} min(A_k, A_{k+1}) - \sum \frac{\emptyset_{i,j,k}^{n} - \emptyset_{i,j,k-1}^{n}}{\Delta z} min(A_{k-1}, A_k) \right) & \end{aligned} \tag{17}$$

For a given Cartesian Grid as shown in Figure 36 such that,

$$A_i = A_{i-1} = A_{i+1},$$

$$A_j = A_{j-1} = A_{j+1},$$

$$A_k = A_{k-1} = A_{k+1}$$

and,

$$\frac{\Omega_{\mathrm{i,j,k}}}{A_i} = \Delta x, \qquad \frac{\Omega_{\mathrm{i,j,k}}}{A_j} = \Delta y, \qquad \frac{\Omega_{\mathrm{i,j,k}}}{A_k} = \Delta z$$

The diffusion equation simplifies to,

$$\emptyset_{i,j,k}^{n+1} = \emptyset_{i,j,k}^{n} + \alpha\Delta t \begin{pmatrix} \left(\frac{\emptyset_{i+1,j,k}^{n} - \emptyset_{i,j,k}^{n}}{\Delta x^2} - \frac{\emptyset_{i,j,k}^{n} - \emptyset_{i-1,j,k}^{n}}{\Delta x^2}\right) \\ + \left(\frac{\emptyset_{i,j+1,k}^{n} - \emptyset_{i,j,k}^{n}}{\Delta y^2} - \frac{\emptyset_{i,j,k}^{n} - \emptyset_{i,j-1,k}^{n}}{\Delta y^2}\right) \\ + \left(\frac{\emptyset_{i,j,k+1}^{n} - \emptyset_{i,j,k}^{n}}{\Delta z^2} - \frac{\emptyset_{i,j,k}^{n} - \emptyset_{i,j,k-1}^{n}}{\Delta z^2}\right) \end{pmatrix} \tag{18}$$

Equation (18) provides the explicit numerical solution for the diffusion equation in the time domain for equation (13).

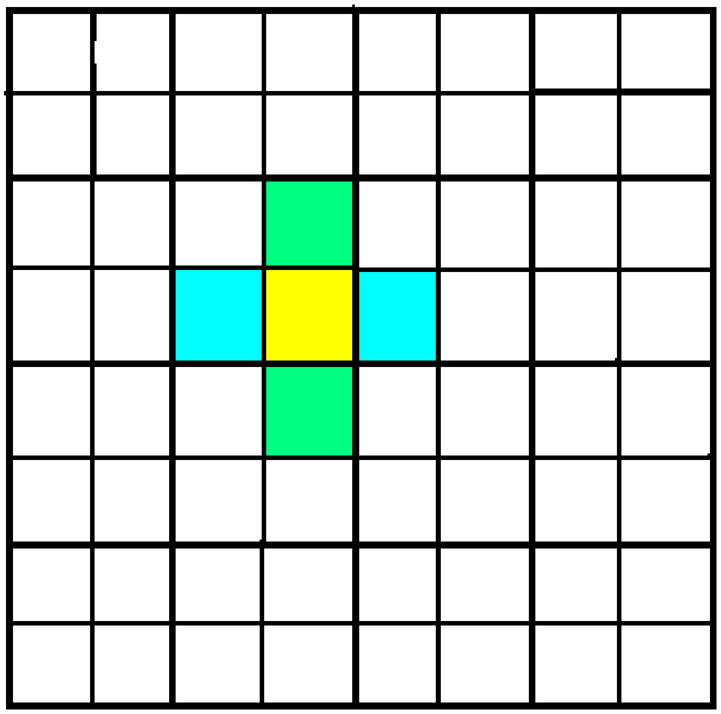


Figure 36: A Cartesian grid in two dimensions

Stability Analysis

Von Neumann's analysis in 3D provides that,

$$\emptyset_{i,j,k}^{n} = e^{at+ik(x+y+z)} \tag{19}$$

Substituting equation (19) into equation (18), one gets

$$e^{at}.e^{ikx}.e^{iky}.e^{ikz}.e^{a\Delta t} = e^{at}.e^{ikx}.e^{iky}.e^{ikz} + \alpha\Delta t\, e^{at}.e^{ikx}.e^{iky}.e^{ikz}\begin{pmatrix} \left(\frac{e^{ik\Delta x}-1}{\Delta x^2}-\frac{1-e^{-ik\Delta x}}{\Delta x^2}\right) \\ +\left(\frac{e^{ik\Delta y}-1}{\Delta y^2}-\frac{1-e^{-ik\Delta y}}{\Delta y^2}\right) \\ +\left(\frac{e^{ik\Delta z}-1}{\Delta z^2}-\frac{1-e^{-ik\Delta z}}{\Delta z^2}\right) \end{pmatrix} \tag{20}$$

Or,

$$e^{a\Delta t} = 1 + \alpha\Delta t\left(\left(\frac{e^{ik\Delta x}-1}{\Delta x^2}-\frac{1-e^{-ik\Delta x}}{\Delta x^2}\right)+\left(\frac{e^{ik\Delta y}-1}{\Delta y^2}-\frac{1-e^{-ik\Delta y}}{\Delta y^2}\right) + \left(\frac{e^{ik\Delta z}-1}{\Delta z^2}-\frac{1-e^{-ik\Delta z}}{\Delta z^2}\right)\right) \tag{21}$$

Using Euler's identities,

$$\cos\theta + i\sin\theta = e^{i\theta}$$

$$\cos\theta - i\sin\theta = e^{-i\theta}$$

A simplified equation can be rewritten as

$$e^{a\Delta t} = 1 + 2\alpha\Delta t\left(\frac{1}{\Delta x^2}(\cos k\Delta x - 1) + \frac{1}{\Delta y^2}(\cos k\Delta y - 1) + \frac{1}{\Delta z^2}(\cos k\Delta z - 1)\right) \tag{22}$$

As,

$$1 - \cos 2\theta = 2\sin^2\theta$$

Hence the equation (22) further simplifies to,

$$e^{\alpha\Delta t} = 1 - 4\alpha\Delta t \left(\frac{1}{\Delta x^2}\sin^2\frac{k\Delta x}{2} + \frac{1}{\Delta y^2}\sin^2\frac{k\Delta y}{2} + \frac{1}{\Delta z^2}\sin^2\frac{k\Delta z}{2}\right) \tag{23}$$

Courant–Friedrichs–Lewy (CFL) condition for the stability of the numerical scheme stipulates that,

$$\left|e^{\alpha\Delta t}\right| < 1 \tag{24}$$

Substituting equation (24) in equations (23) produces,

$$-1 < 1 - 4\alpha\Delta t \left(\frac{1}{\Delta x^2}\sin^2\frac{k\Delta x}{2} + \frac{1}{\Delta y^2}\sin^2\frac{k\Delta y}{2} + \frac{1}{\Delta z^2}\sin^2\frac{k\Delta z}{2}\right) < 1 \tag{25}$$

Or,

$$\alpha\Delta t \left(\frac{1}{\Delta x^2}\sin^2\frac{k\Delta x}{2} + \frac{1}{\Delta y^2}\sin^2\frac{k\Delta y}{2} + \frac{1}{\Delta z^2}\sin^2\frac{k\Delta z}{2}\right) < \frac{1}{2} \tag{26}$$

Since,

$$0 \le \sin^2\theta \le 1 \tag{27}$$

Hence,

$$0 \le \frac{1}{\Delta x^2}\sin^2\frac{k\Delta x}{2} + \frac{1}{\Delta y^2}\sin^2\frac{k\Delta y}{2} + \frac{1}{\Delta z^2}\sin^2\frac{k\Delta z}{2} \le \frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2} \tag{28}$$

Hence the inequality should satisfy the conditions imposed in above equation as well,

$$0 \le \alpha\Delta t \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right) < \frac{1}{2} \tag{29}$$

Now for,

$$\Delta x_{min} = min(\Delta x, \Delta y, \Delta z)$$

$$0 \leq \alpha\Delta t \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right) \leq \frac{3\alpha\Delta t}{\Delta x_{min}{}^2} < \frac{1}{2} \tag{30}$$

Hence the equation (30) will satisfy the Courant–Friedrichs–Lewy (CFL) condition for the stability of the numerical scheme as given below.

$$0 \leq \frac{\alpha\Delta t}{\Delta x_{min}{}^2} < \frac{1}{6} \tag{31}$$

This further imposes the condition on the constant α that,

$$0 < \alpha\Delta t < \frac{\Delta x_{min}{}^2}{6} \tag{32}$$

This also implies that the scheme is numerically stable for a value of α satisfying the above given CFL condition.

In an Octree Grid as shown in Figure 37, if the difference of refinement levels between neighboring cells is enforced to be a maximum of order 1, the possible scenarios could be of the type

$$A_i = A_{i-1} = A_{i+1} = A$$

$$A_i = 2A_{i-1}, \qquad A_i = A_{i+1} = A$$

$$A_i = \frac{1}{2}A_{i-1}, \qquad A_i = A_{i+1} = A$$

$$A_i = \frac{1}{2}A_{i+1}, \qquad A_i = A_{i-1} = A$$

$$A_i = 2A_{i+1}, \qquad A_i = A_{i-1} = A$$

For the scenario a, the diffusion equation for the Octree mesh along the x-axis is the same as that of the Cartesian mesh. Hence the CFL number will also come out to be the same.

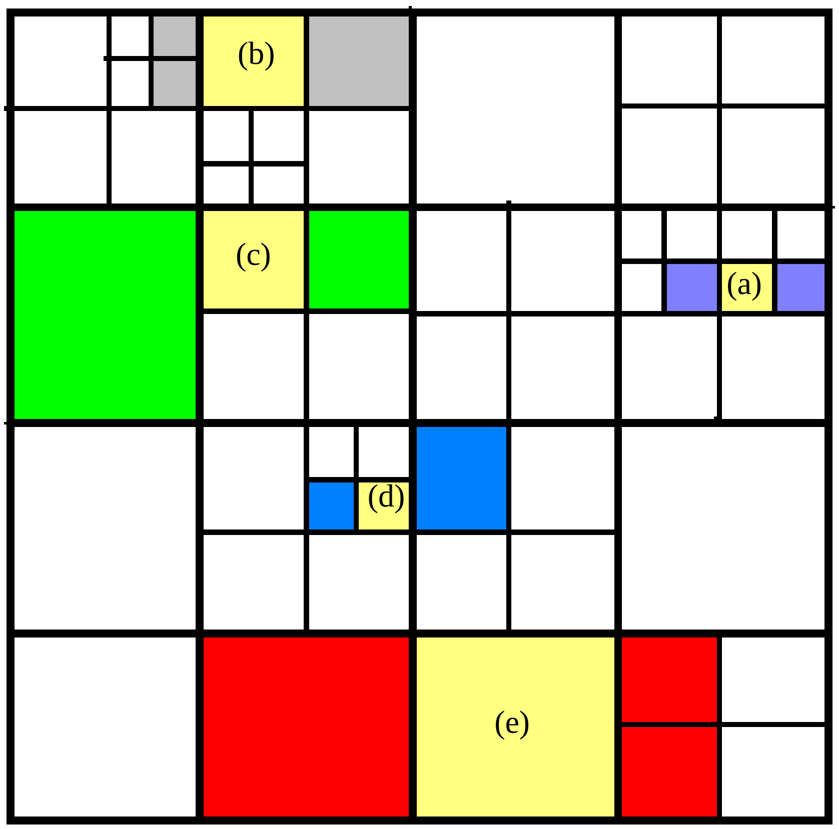


Figure 37: An Octree grid in two dimensions

For scenarios *b* to *d* for an Octree mesh, the numerical scheme for the diffusion equation will be stable if one finds the global minima of Δx_{min} and uses it in the CFL condition for the Cartesian mesh as derived previously. This means that equation (32) for Octree grid could be rewritten as

$$0 < \alpha \Delta t < \frac{\Delta x_{\text{min,global}}^2}{6} \tag{33}$$

Measurement of Change During Solution Process

Sometimes it becomes important to be able to measure the change occurring in each iteration to determine if the numerical solution has reached a desired convergence level. This is achieved by defining a variable $L2norm_{i,j,k}^{n+1}$ at each voxel of the Cartesian grid, where *n* denotes the iteration number, while indices *i*, *j* and *k* represent the location of a voxel within a Cartesian grid.

$$L2norm_{max}^{n+1} = \max\left(\sqrt{\left(\emptyset_{i,j,k}^{n+1} - \emptyset_{i,j,k}^{n}\right)^2}\right) \tag{34}$$

$$L2norm_{avg}^{n+1} = \frac{1}{I \times J \times K}\sum_{\mathrm{i,j,k}}^{\mathrm{I,J,K}}\sqrt{\left(\emptyset_{i,j,k}^{n+1} - \emptyset_{i,j,k}^{n}\right)^2} \tag{35}$$

$$\%\, L2Norm_{change}^{n+1} = \frac{abs\left(L2norm_{avg}^{n+1} - L2norm_{avg}^{n}\right)}{L2norm_{avg}^{n+1}} \times 100.0 \tag{36}$$

Equations (34), (35) and (36) define and quantify three measures of change for successive iterations for the solution of the diffusion equation.

APPENDIX B

COMPUTATIONAL CURVES AND SURFACES

B-Spline Curve: A *p*-th degree B-spline curve is defined by

$$C(u) = \sum_{i-0}^{n} N_{i,p}(u) P_i \qquad \text{where } a \le u \le b \tag{37}$$

where $\{P_i\}$ are control points forming a control polygon, and $\{N_{i,p}(u)\}$ are the *p*-th degree B-spline basis function defined on the nonperiodic knot vector (*m+1* knots)

$$U = \left\{ \underbrace{a,\ldots\ldots a}_{p+1}, u_{p+1}, \ldots\ldots, u_{m-p-1}, \underbrace{b,\ldots\ldots, b}_{p+1} \right\} \tag{38}$$

Unless stated otherwise, *a*=0 and *b*=1. The *i*th B-spline basis function of *p*-degree (order *p*+1), denoted by $N_{i,p}(u)$, is defined as

$$N_{i,0}(u) = \begin{cases} 1 & \text{if} \quad u_i \le u < u_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{39}$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u) \tag{40}$$

where u_i are called knots. The steps required to compute a point on a B-Spline curve at a fixed u value are as follows:

- Find the knot span in which u lies.
- Compute the nonzero basis functions.
- Multiply the values of the nonzero basis functions with the corresponding control points.

NURBS Curve: A pth-degree NURBS curve is defined by

$$C(u) = \frac{\sum_{i=0}^{n} N_{i,p}(u) w_i P_i}{\sum_{i=0}^{n} N_{i,p}(u) w_i} \qquad a \le u \le b \tag{41}$$

where $\{P_i\}$ are control points forming a control polygon, the $\{w_i\}$ are the weights, and $\{N_{i,p}(u)\}$ are the p-th degree B-Spline basis function defined on the nonperiodic and non-uniform knot vector

$$U = \left\{ \underbrace{a,\ldots\ldots a}_{p+1}, u_{p+1}, \ldots\ldots, u_{m-p-1}, \underbrace{b,\ldots\ldots, b}_{p+1} \right\} \tag{42}$$

Unless stated otherwise, $a = 0$ and $b = 1$, and $w_i > 0$ for all i.

NURBS Surface: A NURBS surface of degree p in u direction and degree q in v direction is a bivariate, vector-valued, piecewise rational function of the form

$$C(u) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u) N_{j,q}(v) w_{i,j} P_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u) N_{j,q}(v) w_{i,j}} \qquad 0 \le u, v \le 1 \tag{43}$$

where $\{P_{i,j}\}$ forms a bidirectional control net, the $\{w_{i,j}\}$are the weights, and $\{N_{i,p}(u)\}$ and $\{N_{j,q}(v)\}$ are the non rational B-Spline basis function defined on the knot vector

$$U = \left\{ \underbrace{0,.......0}_{p+1}, u_{p+1}, ..., u_{m-p-1}, \underbrace{1,......,1}_{p+1} \right\} \tag{44}$$

$$V = \left\{ \underbrace{0,.......0}_{q+1}, v_{p+1}, ..., v_{m-q-1}, \underbrace{1,......,1}_{q+1} \right\} \tag{45}$$

Where $r = n + p + 1$ and $s = m + q + 1$.

APPENDIX C

DELAUNAY CRITERIA

Let Z be a point of mesh domain Ω. Considering the Euclidean space defined by (Ω, M(Z)), with $M(Z) = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$, denoted by l^z the distance between two points of Ω in this space. The circumdisc associated with a triangle K, whose center is denoted O^z, is defined in this space by

$$\left(l^z\left(O^z, X\right)\right)^2 = {}^t\overrightarrow{OX} M(Z) \overrightarrow{OX} = k^2 \tag{46}$$

where $X \in R^2$ and k is a real value, such that the disc is circumscribed to triangle K. Hence, the centre O^z is the solution to linear system

$$\begin{cases} l^z\left(O^z, P_1\right) = l^z\left(O^z, P_2\right) \\ l^z\left(O^z, P_1\right) = l^z\left(O^z, P_3\right) \end{cases} \tag{47}$$

and k is precisely $l^z\left(O^z, P_1\right)$. The circumdisc of triangle K encloses the point P, if and only if

$$l^z\left(O^z, P\right) < l^z\left(O^z, P_1\right) \tag{48}$$

In this case, the Delaunay criterion associated with pair (*P*, *K*) is said to be violated according to the metric at point *Z*. By normalizing to one of the above inequalities, a dimensionless measure is defined by

$$\alpha_z(P,K) = \frac{l^z(O^z, P)}{l^z(O^z, P_1)} \tag{49}$$

The violation of Delaunay criterion associated with pair (*P*, *K*) in the metric *Z* means that $\alpha_z(P,K) < 1$.