

GPU BASED PARALLEL SMOOTHING OF SEISMIC TOMOGRAPHY MODELS

IVAN GRIS SEPULVEDA

Department of Computer Science

APPROVED:

Rodrigo Romero, Ph.D., Chair

Olac Fuentes, Ph.D.

Aaron Velasco, Ph.D.

Benjamin C. Flores, Ph.D.
Acting Dean of the Graduate School

Copyright ©

by

Ivan Gris Sepulveda

2011

GPU BASED PARALLEL SMOOTHING OF SEISMIC TOMOGRAPHY
MODELS

by

IVAN GRIS SEPULVEDA, B.S

THESIS

Presented to the Faculty of the Graduate School of
The University of Texas at El Paso
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Department of Computer Science
THE UNIVERSITY OF TEXAS AT EL PASO

December 2011

Acknowledgements

Very special thanks to my mentor Dr. Rodrigo Romero who aided me not only through the development of this document, but throughout my career as well, introducing me to high performance computing, scientific visualization, and to the Cyber-ShARE center, which turned out to be great academic passions in my life. Not only that, but for being a great inspiration for pursuing further studies as a graduate student; I would likely be somewhere else if it weren't for you. I'd also like to thank Dr. Olac Fuentes for his bright ideas and great insights; especially for those intense and great classes in computer graphics and computer vision, and also for his help in the smoothing algorithms research. I'd also like to thank Dr. Aaron Velasco for all his energy and, optimism, and for the time he taught geology for undergraduates four years ago, which inspired me to learn more about the field.

Also, a big thank you to my colleagues Cesar Chacon and Julio Olaya who provided invaluable help with debugging, proofreading, and providing frameworks, tools, tips, tricks, books and moral support.

I also want to thank my other colleagues at the Cyber-ShARE Center of Excellence who supported me in many other ways, Javier Garcia, Mary Contreras, Patty Esparza, Leo Salayandia, Liz Pardo and Valeria Estrada.

Last, but by no means least, I'd like to thank my family who has supported me and continues to do so in every describable way and to my friends Salvador Ruiz, Noel Reza, Esthela Gallardo, Laura Caballero, Dinora Velarde, Pablo Cervantes, and Abril Holguin who helped me stay sane during the process.

Thank you.

This material is based upon work supported in part by the National Science Foundation under CREST Grant No. HRD-0734825 and Grant No. CNS-0923442. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

Abstract

Abstract

Three-dimensional models of the velocity structure of the Earth's crust are an important and relevant factor for several types of analyses across disciplines. Crustal velocity models are also commonly used to analyze and search for different materials of interest or to determine and differentiate many aspects of life on Earth during different eras.

Seismic tomography techniques, both in two and three dimensions, perform image reconstruction of the crust of the Earth [10]. Seismic tomography algorithms can calculate crustal velocity structure through inversion of traveltimes of seismic waves produced by natural events, such as earthquakes, or controlled source experiments, such as explosions.

The work presented in this thesis is based on the utilization of tomographic data sets produced through controlled-source experiments and the application of an iterative first-arrival traveltime seismic tomography algorithm ("STA") to obtain 3D velocity models of specific regions of the crust of the Earth.

The research focus of this thesis is to identify and exploit potentially parallelizable functions of smoothing algorithms, which are the STA performance bottleneck. Parallel tasks are then mapped to the architecture of a graphics processing unit ("GPU") to accelerate the smoothing execution speed while maintaining the consistency and reliability of the outputs with respect to models considered as correct reference outputs. The implemented parallel STA smoothing algorithms deliver peak performance improvements from 31.9% to 73.1% and average improvements from 20.9% to 66.4% with respect to the fastest sequential implementations of the algorithms.

Table of Contents

Acknowledgements.....	v
Abstract	vi
Table of Contents.....	vii
List of Tables.....	ix
List of Figures	x
Chapter 1: Introduction	1
1.1 Goals.....	2
Chapter 2: Background	3
2.1 Seismic Traveltime Tomography	3
2.2 Traveltime Residuals and Model Convergence	11
Chapter 3: Smoothing Algorithms.....	12
3.1 Smoothing Filters	12
3.2 STA Smoothing.....	14
3.3 Basic STA Smoothing Algorithms.....	15
3.4 Caching Smoothing Algorithms.....	17
3.5 Eliminating Redundancy while Maintaining Smoothing Parallelism ..	19
3.6 Summed Volume Tables.....	21
3.7 Velocity Perturbation Smoothing with Summed Volume Tables.....	22
Chapter 4: STA Smoothing Algorithms Analysis for Parallelization	24
4.1 Parallel Algorithm Design	24
4.2 GPU Architecture.....	27
4.3 Summed Volume Tables for Parallel STA Smoothers.....	30
Chapter 5: Results.....	32
5.1 Experimental Settings and Data Set	32
5.2 Cell Smoother Performance Analysis	33
5.3 GPU Performance Analysis of STA Cell Smoothing Algorithm	38
5.4 Performance Analysis of STA Vertex Algorithm	43
5.5 GPU Performance Analysis of the STA Vertex Smoothing Algorithm.....	47

Chapter 6: Conclusion	Error! Bookmark not defined.
6.1 Future Work	Error! Bookmark not defined.
References	54
Vita	56

List of Tables

Table 1: Computation time per iteration	15
Table 2: Cell smoother partitioning	24
Table 3: Vertex smoother partitioning	24
Table 4: NVIDIA QUADRO 5000 GPU Device Specifications	29
Table 5: Execution times in milliseconds per iteration of the STA cell smoothing algorithms	37
Table 6: GPU time breakdown for the cell smoother	39
Table 7: GPU time breakdown for kernels, memory transfers and occupancy for the cell smoother	39
Table 8: Occupancy analysis for cell smoothing kernel	40
Table 9: Memory throughput analysis for cell smoothing kernel on device Quadro 5000	42
Table 10: Analysis for cell smoothing kernel on device Quadro 5000.	42
Table 11: Performance of the vertex smoothing algorithms	46
Table 12: GPU time breakdown for the vertex smoother	48
Table 13: GPU time breakdown for kernels, memory transfers and occupancy for the vertex smoother	48
Table 14: Occupancy analysis for the vertex smoothing kernel	49
Table 15: Memory throughput analysis for vertex smoothing kernel on device Quadro 5000	50
Table 16: Analysis for the vertex smoothing kernel on device Quadro 5000	51

List of Figures

Figure 1: Seismic traveltime tomography algorithms	5
Figure 2. 1D velocity model.....	6
Figure 3: 3D velocity model generated from a 1D velocity model.....	6
Figure 4. Discrete first arrival time 3D model, where the red region shows the smallest times found in the neighborhood of the shot point and the blue region shows the longest times	7
Figure 5: Vector coverage file in XML format, containing shotpoint location, receiver location, pick time, and ray trajectory as a list of vertices.....	8
Figure 6: Pixel ray coverage, where the darkest regions represent higher ray density per cell	9
Figure 7: Velocity perturbations computed for the velocity model.....	9
Figure 8: Sample smoothing schedule	10
Figure 9: Three dimensional velocity model after several iterations	11
Figure 10: Example of color smoothing, also known as blur	13
Figure 11: Example of 3D model smoothing	13
Figure 12: Overlap of smoothing volumes for two elements along the X-axis using the basic STA smoothing algorithm.....	16
Figure 13: Overlap of smoothing volumes for two elements along the X-axis using the basic STA smoothing algorithm.....	17
Figure 14. Overlapping smoothing volumes showing cached slices (green region) that are useful for computing both volumes.	18
Figure 15. Smoothing volumes overlapping in the Z direction. The green region represents cached data along the Z-axis	19
Figure 18: Calculating the sum of volumetric properties of a rectangular prism using a summed volume table.....	22
Figure 20: Data Dependence Graphs of STA Smoothers	25
Figure 21: Potential data transfers between the CPU (host) and the GPU (device).....	26
Figure : GPU memory structure [17].....	28
Figure 23: Execution time in milliseconds per iteration for the different versions of the cell smoother ..	34
Figure 24: Execution times for the cell smoother with an emphasis on the caching technique, which is plotted in green.....	35

Figure 25: Execution time comparison for the parallel and sequential versions of the SVT cell smoothing algorithm	36
Figure 26: Percentage of improvement per iteration per algorithm compared with the basic cell smoother	37
Figure 27: GPU time summary plot for the cell smoother.....	39
Figure 28: With plot with overlapping CPU and GPU times for the cell smoother.....	40
Figure 29: Execution time in milliseconds per iteration for the different versions of the vertex smoother	44
Figure 30: Execution times for the vertex smoother with an emphasis on the caching technique, which is plotted in maroon.....	44
Figure 31: Execution time comparison for the parallel and sequential versions of the SVT vertex smoothing algorithm	45
Figure 32: Percentage of improvement per iteration per algorithm compared with the basic vertex smoother.....	46
Figure 33: GPU time summary plot for the vertex smoother.....	47
Figure 34: With plot with overlapping CPU and GPU times for the vertex smoother.....	49

Chapter 1: Introduction

Three-dimensional information about the velocity structure of the Earth's crust is an important and relevant factor for several types of analyses across disciplines. This information can be used to calculate the environmental impact of human actions and ground factors affecting the construction of buildings and other large structures. Crustal information is presented in models that are also commonly used to analyze and search for different materials of interest including oil, water, and certain minerals. Analyzing the structure of the crust of the Earth in three-dimensional models can also help to determine and differentiate many aspects of life on Earth during different eras and, in some cases, even aid in the prediction of the possible behavior and activity of the Earth along certain features such as faults, plates, and volcanoes. Models of these three-dimensional structures can be obtained through the inversion of seismic traveltimes [10].

Tomography in geological terms is the two- or three-dimensional, 2D and 3D, respectively, image reconstruction of the crust of the Earth [10]. Seismic tomography can be performed through algorithms that perform seismic traveltime inversion based on seismic waves produced by natural events, such as earthquakes, or controlled environments, such as explosions.

The work presented in this thesis is based on the utilization of controlled environment experiments and the application of an iterative first-arrival traveltime seismic tomography algorithm ("STA") to obtain 3D models of specific regions of the crust of the Earth. While the operations of the algorithm have been enhanced through the years since its creation in 1988, this work is focused on smoothing operations because they are the most computationally intensive and time consuming of all the operations comprising the algorithm. Both performance optimization opportunities inherent in the current sequential smoothers, referred to as the "basic smoothers" in this document, and alternative parallel smoothers will be considered.

The research focus of this thesis is to identify and exploit potentially parallelizable functions of the smoothing algorithms. Then map parallel tasks to the capabilities of a graphics processing unit ("GPU") to accelerate the smoothing execution speed while verifying the consistency and reliability of the outputs with respect to models computed with the current sequential implementation of the STA which will be considered as correct reference outputs.

1.1 Goals

The following goals lay the foundation for this thesis:

1. To perform a parallelization analysis of current sequential smoothing algorithms
2. To design, map, and implement parallelizable operations of such algorithms

The first goal will be achieved by analysis of three variations of the two types of smoothing algorithm of Hole-Vidale's STA. The second goal is attained by following Foster's method [7] to analyze, partition, agglomerate, and map algorithmic tasks. Parallel algorithm design is based on primitive tasks operating on summed area table ("SAT") algorithms which used for texture mapping, face detection, real time rendering of glossy environmental reflections, blurriness, glossy transparent object rendering, depth of field rendering, empty space removal, ambient occlusion and halo generation in computer graphics and for face detection in computer vision applications [5] [6] [9] [13] [18] [22] [26]. Volume processing extensions of the SAT algorithms are adapted in this thesis for geophysical structure reconstruction. Algorithm tasks are mapped to a heterogeneous CPU/GPU architecture. Parallel tasks are implemented with the CUDA programming environment [17] and executed in an NVIDIA Fermi GPU [18].

Chapter 2: Background

In order to understand the proposed smoothing optimizations and the architecture of the solution, it is necessary to analyze the seismic tomography algorithm. Since it is an iterative process and the different modules are dependent, it is important to understand the flow, the purpose and the way the experiments and models are created to grasp the improvements made to the algorithm as a whole, and to understand the innovative approach in using graphical processing units for seismic tomography computations.

In the following section, the different modules of the seismic travel time tomography algorithm are described along with the optimizations made by geologists several years ago. In addition, I explain the different ways in which this algorithm can be applied and the possibilities, benefits and disadvantages of creating models with different measurement techniques. Finally, I also go in detail explaining several seismic tomography concepts and the key differences between this algorithm and other seismic tomography algorithms.

After the background is set, the next section focuses on the parts of the algorithm that are parallelized, and some approaches taken in different disciplines for similar algorithms.

2.1 Seismic Traveltime Tomography

The first-arrival seismic traveltime 3D tomography algorithm used as the basis for this work was designed and implemented by John A. Hole [11] using John Vidale's algorithm and implementation for forward modeling [25]. Tomograph means “slice picture”. Geophysicists use seismic tomography to describe two- and three-dimensional imaging. A first-arrival traveltime algorithm predicts, i.e., computes, Earth's structure by computing the velocity of seismic waves generated by a source such as an earthquake, an air gun, or an explosion and the time that it takes for the seismic wave front to reach a set of sensors such as seismographs and geophones.

Hole's STA uses forward and inverse modeling procedures to provide a major speed improvement with respect to previous algorithms which are based on costly ray tracing. Forward modeling uses Vidale's 2D and 3D algorithms for rapid and accurate computation of discrete first arrival traveltimes with finite differences and the eikonal equation [25]. After forward modeling, back tracing of seismic wave ray paths and inversion are used to

the compute velocity misfits needed to make traveltimes residuals, i.e., the differences between observed and calculated first arrival times, and other convergence criteria approach their acceptance thresholds.

Vidale-Hole's forward and inverse modeling techniques allowed dense model sampling, which in turn produces high-resolution discrete models [10][22]. Typical models are grids divided into pixels or voxels, for 2D and 3D models, respectively, with edge lengths in the order of 1 km, which is a relatively small measure in geological terms when compared to the volume of the Earth. Such discrete models can be quite large and contain several hundreds of thousands or even millions of pixels or voxels, which are referred to as cells in general.

Besides allowing high resolution in terms of grid size, also Vidale-Hole's techniques also allow dense sampling, which means that they allow many ray paths generated by the trajectory of the primary waves to sample any cell. That is, hundreds of rays can contribute to the pool of structural information of a single cell. The combination of these algorithmic features produces spatially well resolved 3D tomographic images.

Hole's STA also improved upon other algorithms by handling large lateral velocity variations and having stability even in the presence of noisy data. Seismic noise is introduced by random sources near the sensors such as animal footsteps or urban activities. Recorded seismic noise may mislead the analyst by obfuscating the reading of first arrival times of experimental seismic waves.

Previous tomographic algorithms have mainly two limitations. The first one was caused by the nonlinearity of the inversion. Many tomographic inversion techniques avoid the fact that the ray paths depend on the unknown structure by assuming that the velocity variations are negligible and ray paths are stable. However, both accurate forward modeling (three-dimensional two-point ray tracing) and linear inversions are very slow computational processes [10]. The second limitation was that the computational costs limit the spatial resolution of the model when inversion requires the solution of a system of linear equations that relate traveltimes to model parameters [10].

The STA is implemented by a set of executable modules. Module inputs and outputs are based on data and argument files. This data flow enables visualization and analysis of each computed model. In the following sections, the most relevant STA modules are briefly described together with a short description of the forward and inverse modeling techniques.

2.1.1 Algorithm Overview

```
Create or read initial velocity model
While (iterative step needed) do
    For (each source) do
        Compute first arrival time model
    For (each receiver and source) do
        Compute ray coverage
        Compute velocity perturbation
    End of for
    Velocity perturbation cell smoothing
    Velocity perturbation vertex smoothing
    Update velocity model
End of while
Output velocity model
```

Figure 1: Seismic traveltime tomography algorithms

The STA implementation is based on the creation and manipulation of discrete three-dimensional models of velocity, traveltimes, and velocity perturbations. Briefly described, the algorithm performs a series of iterations that update the velocity model with velocity perturbations which reduce the traveltime residuals until some convergence criteria are met. Depending on the computation, the description of a step may refer to slowness, the inverse of velocity, for expediency. The key points of the algorithm that will be analyzed, optimized, and parallelized for GPUs are the smoothing steps which are moving average filters that provide model stabilization and convergence.

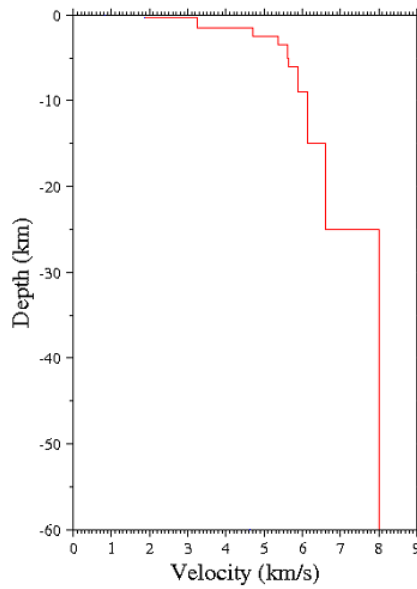


Figure 2: 1D velocity model

2.1.2 Velocity Model

The initial 3D discrete velocity model required by the STA can be generated from an estimate interpretation of a one dimensional velocity model – a mapping of velocities vs. depths (Fig 2). Such a mapping is extrapolated along an x axis and a y axis to form a 3D model. The separation between the discrete model samples, which are referred to as vertices, is uniform for Hole's STA, but in general can be set to a different value along each axis depending on the employed algorithm and the needed output resolution.

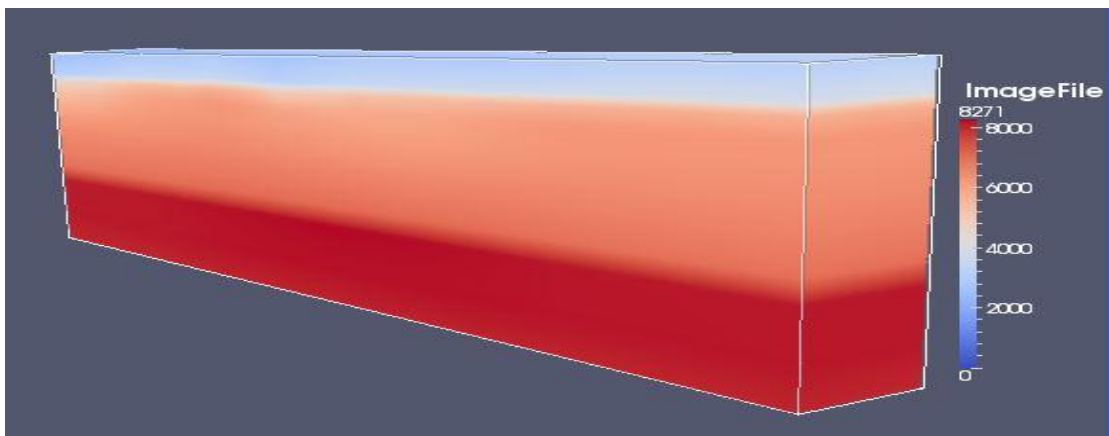


Figure 3: 3D velocity model generated from a 1D velocity model

2.1.3 Ray Coverage and First Arrival Times

Using the discrete 3D velocity model, the STA computes a discrete 3D time model comprised of the first arrival traveltimes for each vertex in the model. Calculated first arrival times are stored on a file per shotpoint. The shots and receivers can be located anywhere within the model.

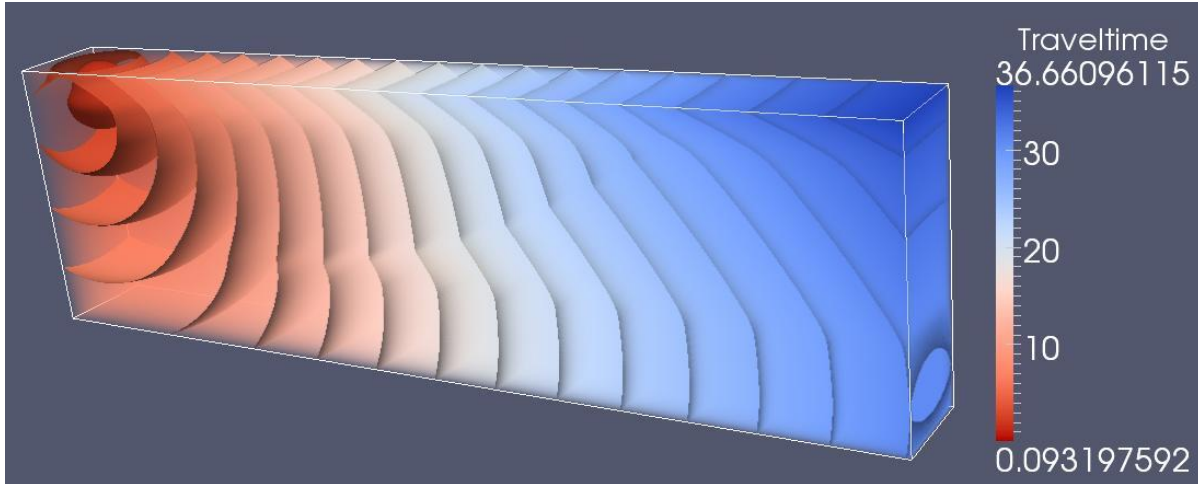


Figure 4: Discrete first arrival time 3D model, where the red region shows the smallest times found in the neighborhood of the shot point and the blue region shows the longest times

Ray paths are found by back tracing from the receiver locations to the shot point location through the computed traveltime field of each shot point. Within each model cell, ray path segments are assumed to be straight and directed against the travel time gradient across the cell. Because of this, ray back tracking is very fast and efficient. The computation time for each shot point depends approximately linearly on the number of rays and the length of each ray.

```

1  <ray_coverage>
2
3  <ray   line="          1   "
4        shot="          1002  "
5        shot_x="  0.0000000000000000  "
6        shot_y="  0.0000000000000000  "
7        shot_z="  6.6990000000000000  "
8        rec_x=" -57152.000000000000  "
9        rec_y="  857.00000000000000  "
10       rec_z="  6667.000000000000  "
11       pick_time=" 10.36014000000000  " >
12
13  <vertex x=" -57.15200000000000  "
14        y="  0.8570000000000000  "
15        z="  6.6670000000000000  " >
16  </vertex>
17  ----- Additional ray vertices go here -----
18
19
20  </pe>
21
22  ----- Additional rays go here -----
23

```

Figure 5: Vector coverage file in XML format, containing shotpoint location, receiver location, pick time, and ray trajectory as a list of vertices

Vidale's 3D finite difference traveltimes algorithm is used to compute first arrival traveltimes to each vertex in the model [22]. The algorithm uses finite difference operators based on the eikonal equation to calculate the first arrival times of direct, refracted, diffracted, and head waves. Utilized finite difference operators are a function of the average slowness across model cells, which is equivalent to approximating a seismic wave front as a half-space within each cell. The discrete time and models are sampled at a set of uniformly spaced grid points in three dimensions.

Ray coverage, which is used for velocity perturbation smoothing and final velocity structure computation and quality assessment, is calculated as the sum of ray paths that cross each cell.

2.1.4 Velocity Perturbation

The model velocity perturbations, equivalently handled as slowness perturbations, solve the linear inversion problem. To solve a linearized version of the true non-linear traveltimes tomography problem, iterations are required. The slowness model is updated by the addition of perturbations and is then used as the new reference model for the next linearized inversion iteration. STA iterations are stopped when some criteria are satisfied; in

this work, the root mean square traveltimes residuals must fall below the experimental error estimated by the analyst.

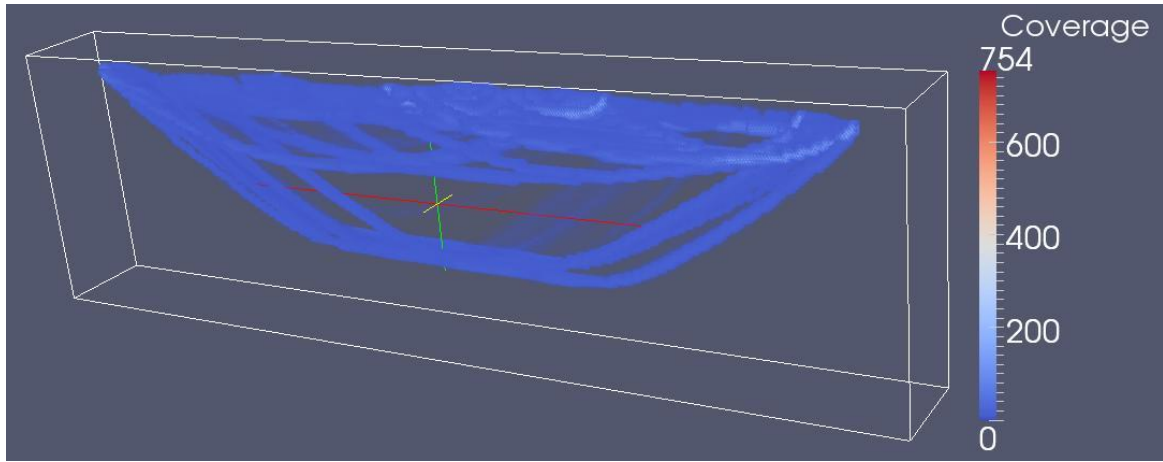


Figure 6: Pixel ray coverage, where the darkest regions represent higher ray density per cell

For every back traced ray path, slowness is computed and added to model cells traversed by the ray and each traversed cell ray count is incremented also. Once all rays have been traced, the slowness perturbation and the ray coverage of the model are used for smoothing the slowness perturbations before using them to update the velocity model.

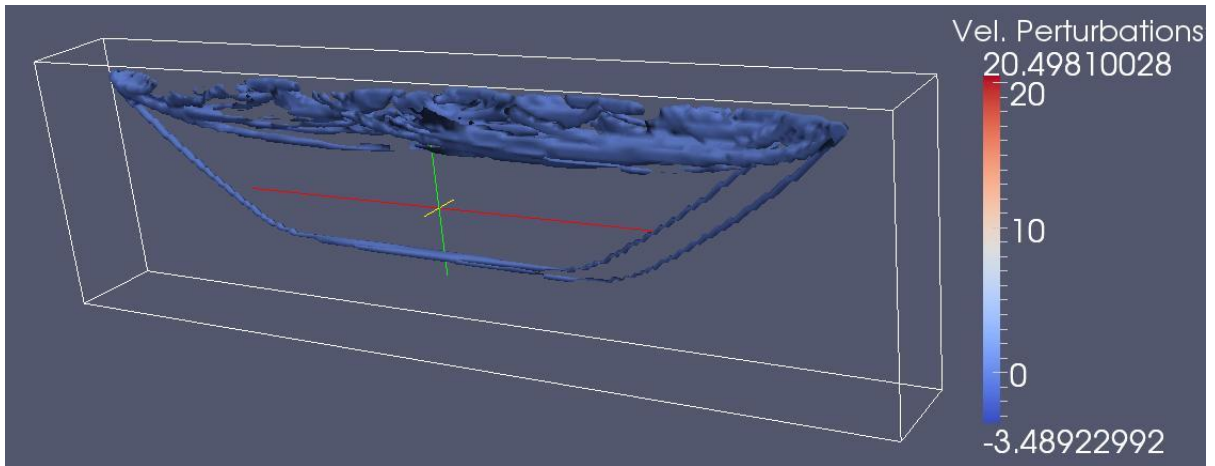


Figure 7: Velocity perturbations computed for the velocity model

2.1.5 Velocity Perturbation Smoothing

Before updating the velocity model in a given iteration, calculated velocity perturbations and ray coverage are utilized for cell perturbation smoothing and vertex perturbation smoothing. The STA smoothing parameters specify the size, the sequence, and the number of passes of each of the smoothers. This set of parameters is referred to as the smoothing schedule. The smoothing filter size defines the number of cells or vertices to include in each dimension of a smoothing volume that is averaged to soften velocity perturbations before applying them to the velocity model. Smoothing helps to avoid computing model representations with physically impossible velocities or unlikely interfaces [1].

```
set smthx=( 96 48 24 12 6 4 )
set smthy=( 24 24 12 8 4 2 )
set smthz=( 24 16 8 6 2 2 )

set mvax=( 97 49 25 13 7 5 )
set mvay=( 25 25 13 9 5 3 )
set mvaz=( 25 17 9 7 3 3 )
```

Figure 8: Sample smoothing schedule

The cell smoother adds all the cell slowness perturbations within the smoothing volume and divides the total by the sum of rays within the same volume. This means that the only affected cells in the entire model are those with a nonzero coverage in their smoothing volume. Notice that the smoothing volume decreases in size with each iteration to resolve the smallest model features for a given cell size. The smoothed velocity perturbations computed by the cell smoother are applied to the vertex that is closest to the model origin in each cell. Unlike the cell smoother, the vertex smoother computes the arithmetic average of the velocity perturbations of all the vertices in the smoothing volume and adds the result to the corresponding vertex in the velocity model.

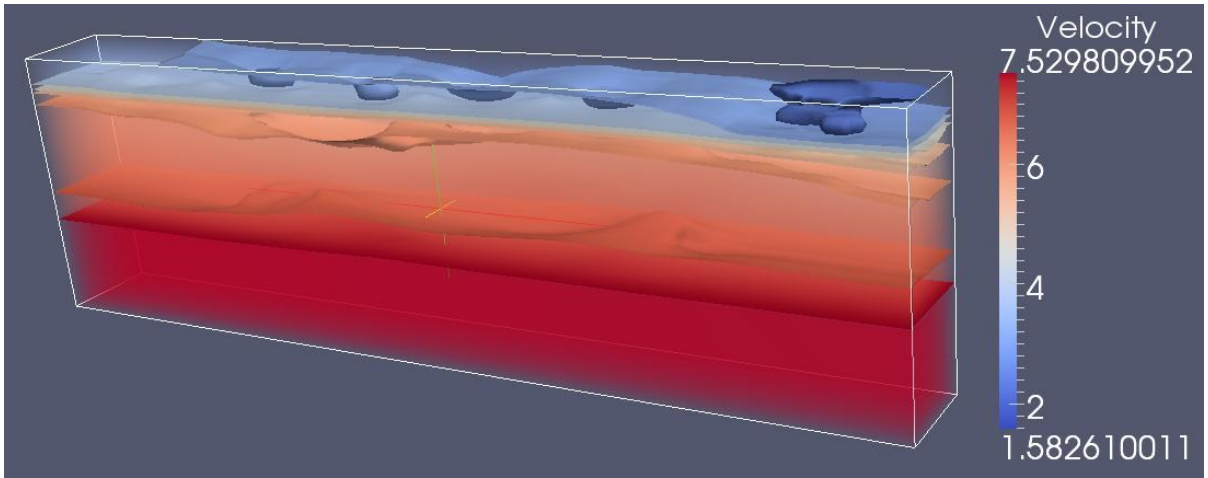


Figure 9: Three dimensional velocity model after several iterations

2.2 Traveltime Residuals and Model Convergence

A model is said to converge once all ray paths have been traced from shot-point locations to receiver locations, or vice versa due to algorithmic symmetry, and its root mean square (“RMS”) traveltime residuals, i.e., the differences between observed and calculated traveltimes, are less than the traveltime measurement error estimated and deemed acceptable by the analyst. The model overall RMS residual is calculated using the traveltime misfits of all shot-receiver pairs of the experiment. It is unlikely that a calculated ray path is exact, meaning that it has a zero residual, mainly because of tomographic model non-uniqueness [11], observed traveltime measurement uncertainty [2], linear approximations for solving a non-linear problem [11], and numerical errors introduced by rounding, truncation, and inaccurate representation of real numbers using machine floating point data types. Each traveltime residual, computed in seconds as the difference between the calculated arrival time and the observed traveltime, is evenly distributed to all the cells that the ray path traverses. In a converging model, the RMS ray path residuals will tend to decrease and eventually fall below the acceptable RMS threshold, which indicates that the velocity model is acceptably accurate, i.e., no further iterations are required, and can be used for a geophysical analysis and interpretation of the velocity structure of the modeled region.

Chapter 3: Smoothing Algorithms

Smoothing, which is also referred to as smudging, blurring, smearing, softening, and filtering depending on the field of application, is the process of distributing input values through a greater neighboring area or volume than the one associated with the input by degrading them depending on their distance from a reference location and combining them according to a weight function with existing or newly calculated neighboring values. There are several ways in which smoothing can be applied. In the following sections, the STA basic smoothing algorithms are described and analyzed. In addition, applications of smoothing in computer graphics are discussed and the parallelization and mapping to GPU architectures of a new STA smoothing algorithm is explained.

3.1 Smoothing Filters

Smoothing is a filtering operation that is commonly used in computer graphics as a way to distribute texture details or color values of a certain amount of pixels in a specific zone of a rendered image to simulate motion blur or reduce the visual impact of underlying polygonal representations, respectively, or as a mask that highlights specific features to an existing image, such as edges or shadows, to enhance depth and shape perception.

Three-dimensional filters for seismic tomography smoothing are different from two-dimensional image filters found in computer graphics in the following aspects:

1. Filters for images and 3D surfaces are planar filters that cannot be applied to a 3D model surface without producing a distorting anomaly in the model. In contrast, the STA smoothing filter affects a 3D volume not just an area or surface of the model. Smoothing of 3D surfaces in computer graphics (Fig. 11) may add detail to an existing model, for instance, by generating additional polygonal subdivisions or triangulations that alter the vertex and polygonal count of the surface to simulate

continuity and curves. However, the STA smoothing volume filters do not generate additional geometry; instead, the STA filters attempt to distribute velocity misfit information to adjacent cells or vertices.

2. Filtering for computer graphics can be localized, as shown on the second vase in Fig. 11, or distributed, as shown on the third vase in Fig. 11. In addition, region size and shape might also vary. However, the STA cell and vertex smoothers are always distributed and use a constant volume per iteration.



Figure 10: Example of color smoothing, also known as blur

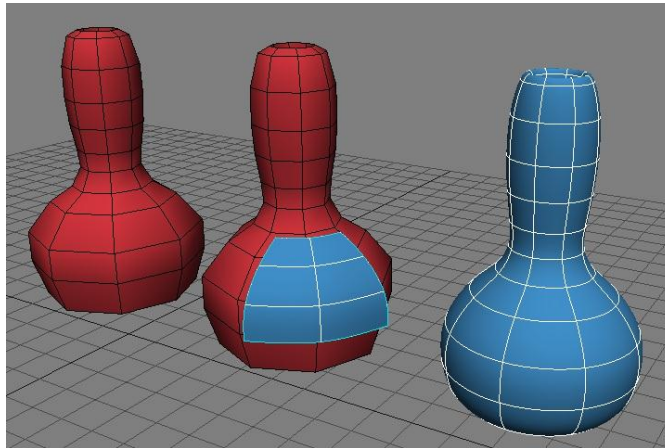


Figure 11: Example of 3D model smoothing

3.2 STA Smoothing

The main purpose of the STA smoothers is to quickly stabilize the model. With the presence of noise, the conversion from a 1D model to a 3D velocity model extends the noise to the two additional dimensions and can generate artifacts. Since the STA is iterative, if the model does not stabilize early in the process, the impact of noise and other errors can be propagated through iterations and turn into irresolvable artifacts that prevent the model from ever converging. When applying a large smoothing pass, those undesirable features are either lost or significantly reduced by sparse distribution throughout the model, i.e., smudging the model removes or makes the noisy features undistinguishable. However, reducing the noise also unknowingly removes fine features of the model that may occupy only a few cells, which will cause a resolution loss in the model.

On most of a seismic tomography velocity model, it is assumed that neighboring cells are of the same material and changes occur gradually (although the STA can handle large velocity contrasts). Thus model smoothing uses this assumption and spreads material properties from a cell to its neighbors.

To maintain stability while enhancing model resolution, smoothing volumes change per iteration. Since large volumes will tend to stabilize the model and enhance the likelihood of convergence and small volumes will tend to highlight small features, the STA combines large smoothing volumes in initial iterations with small smoothing volumes in the last iterations, gradually reducing volumes to achieve early stability and late resolution.

Both the cell smoother and the vertex smoother of the STA are computationally expensive. Table 1 provides an illustrative comparison of the execution times of the main steps of the STA for a model with 414414 vertices, 7 shot points, and 793 receivers, using the smoothing schedule in Fig. 8. Notice that both smoothers will sequentially process information from all model cells, even those with a null coverage or a null velocity perturbation. These cells and their vertices are relevant only if their smoothing volumes contain any cell or vertex with a positive coverage or a non-zero velocity

perturbation. Thus, in addition to being the most computational intensive and time-consuming STA processes, both smoothers are impacted by the overhead needed to process irrelevant cells or vertices of the model. The basic algorithm is explained in the next section.

Table 1: Computation time per iteration

Computations	Smoothing schedule	Time per Iteration
First arrival time	N/A	0.248s
Ray coverage	N/A	0.054s
Cell smoothing	96 x 24 x 24	113.716s
Cell smoothing	48 x 24 x 16	41.541s
Cell smoothing	24 x 12 x 8	6.332s
Cell smoothing	12 x 8 x 6	1.718s
Cell smoothing	6 x 4 x 2	0.170s
Cell smoothing	4 x 2 x 2	0.064s
Vertex smoothing	97 x 25 x 25	97.731s
Vertex smoothing	49 x 25 x 17	36.235s
Vertex smoothing	25 x 13 x 9	6.240s
Vertex smoothing	13 x 9 x 7	1.914s
Vertex smoothing	7 x 5 x 3	0.292s
Vertex smoothing	5 x 3 x 3	0.153s

3.3 Basic STA Smoothing Algorithms

The basic algorithms, which were originally implemented in Fortran, address the smoothing problem utilizing a vast amount of calculations. Both the cell smoother and the vertex smoother are based on the computation of a moving average filter that essentially is highly redundant. While the former averages only cells with a positive coverage, the latter averages all vertexes in the smoothing volume.

In both smoothers, the algorithm processes all the elements, which are either cells or vertices, of the input 3D seismic tomography model by calculating the limits of a smoothing volume with the element in the center but clipped against the model. Then the smoothing volume is traversed to compute a summation of all the element values. Finally, the summation result is divided, depending on the

smoother, either by the accumulated coverage or the number of cells contained within the smoothing volume to produce a new, smoothed value of the element being processed.

The major inefficiency of the basic algorithm is that as the distributed filter traverses the model, computations for each smoothing volume are independently and redundantly performed from those of other smoothing volumes. Due to poor locality of reference, memory accesses are costly, but summation results that had been partially solved have to be recomputed for each smoothing volume.

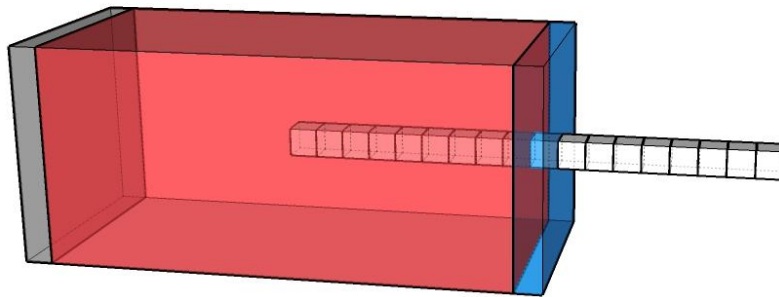


Figure 12: Overlap of smoothing volumes for two elements along the X-axis using the basic STA smoothing algorithm

This situation is illustrated in Fig. 12 using part of a row to be processed along the X-axis direction and two, mostly overlapping, smoothing volumes. The elements at the center of the smoothing volumes are the two leftmost elements in the row to process. The red region indicates redundant memory accesses and associated calculations which are performed by the basic smoothers as the filtering moves from the element on the left to the element on the right traversing a line of the model in the X axis. Fig. 13 illustrates filter displacement along the Z axis. Notice that the STA uses a left-handed coordinate system with a positive Z axis representing increasing depth in the Earth's crust. To reduce the redundant memory accesses and calculations, caching was introduced in the basic algorithm. Caching

produces a significant performance improvement with respect to the basic non-caching implementation in both smoothers [19.1].

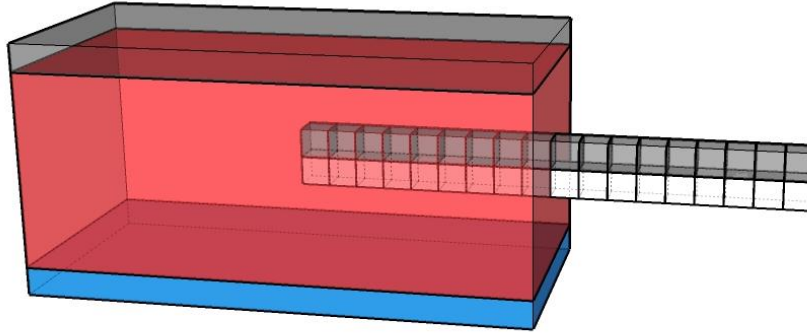


Figure 13: Overlap of smoothing volumes for two elements along the X-axis using the basic STA smoothing algorithm

3.4 Caching Smoothing Algorithms

The caching smoothing algorithms store smoothing volume computations in a slice result array. The smoothing volume is traversed along the smoothing direction to compute orthogonal slices of the volume. Since the STA displaces the smoothing volume in the X direction, orthogonal slices have the thickness of one element in the X direction and extend in the Y and Z direction to the end of the smoothing volume. Most of the cached data can be reused to eliminate redundant processing within a set of contiguous smoothing volumes in the smoothing direction. When the smoothing volume moves, depending on clipping for the previous position and the current position, the first slice for the previous volume may not be considered for the current volume and a new last slice may have to be computed for the current volume. This algorithmic modification significantly reduces redundancy of memory accesses and computations to process smoothing volumes as shown in Fig.14. However, the modification of the algorithm is very complicated to implement for more than one axis. Thus, when the

smoothing volume moves in either the Y or the Z direction, the cache is flushed and redundancy is unavoidable.

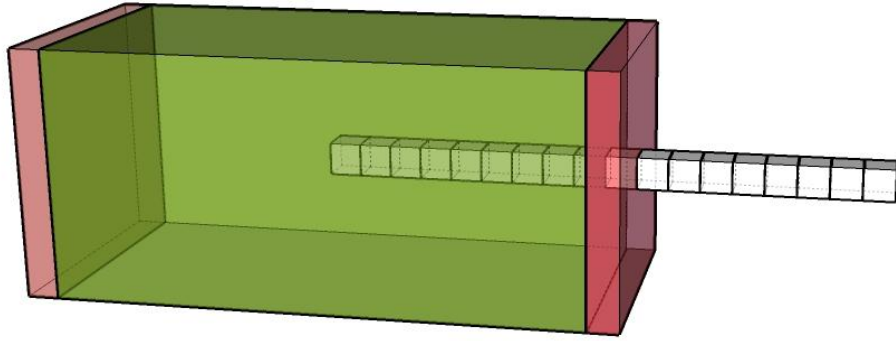


Figure 14: Overlapping smoothing volumes showing cached slices (green region) that are useful for computing both volumes

To remove most redundancy in the three dimensions instead of just in one as explained above, a three-dimensional cache would be needed as shown in Fig. 15. The operations needed to maintain the cache along each dimension would make implementation very complex and error prone. Most importantly, the smoothing volume dependences implicit in a 3D cache would eliminate the possibility of parallelization of the smoothing algorithms as an additional strategy to improve STA performance.

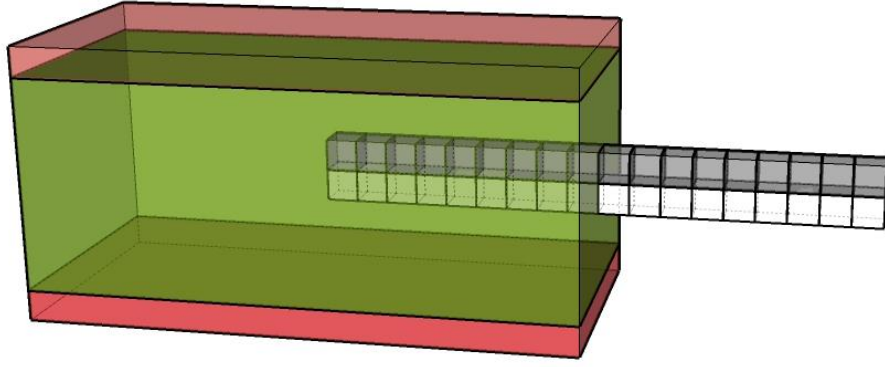


Figure 15. Smoothing volumes overlapping in the Z direction. The green region represents cached data along the Z-axis

3.5 Eliminating Redundancy while Maintaining Smoothing Parallelism

To remove the impact of redundancy of memory access and computations combined with data dependencies of both the STA basic smoothers and the caching smoothers introduced by Olaya et al [19.1] a new algorithm was developed by Romero and Fuentes [23] using a summed table technique introduced by Crow [5] for use in computer graphics and later used by Viola and Jones [26] for computer vision. This thesis parallelizes the summed table smoother and maps it to a GPU architecture as discussed in the following chapters. This section and the next discuss summed table techniques as a reference for the parallelization and mapping analyses presented in the following chapters.

Crow presented an algorithm [5] to compute the sum of any rectangular subset of elements in a grid in constant time, which was later applied by P. Viola and M. J. Jones [26] for face detection algorithms. The technique is also applicable to compute the sum of two or more non-contiguous subsets of a grid and it can be upgraded to higher dimensions. Romero and Fuentes [23] applied the technique for three-dimensional seismic velocity models.

3.5.1 Summed Area Tables

Rectangular two dimensional image features can be computed rapidly using an intermediate representation referred to as a summed area table (SAT) by Crow [5] and the integral image by Viola and Jones [5]. Referring to the input as $i(x,y)$ as the input image, the summed area table $sat(x,y)$ is the sum of the pixel values for all pixels at equal or lower x and y positions, i.e.,

$$sat(x, y) = \sum_{\substack{x' \leq x \\ y' \leq y}} i(x', y')$$

As an computational enhancement, $sat(x,y)$ can be computed in a single pass using the following equations:

$$rs(x, y) = rs(x, y - 1) + i(x, y)$$

$$sat(x, y) = sat(x, y - 1) + rs(x, y)$$

where $rs(x, -1)$ and $sat(-1, y)$ are both defined as zero.

Once the SAT has been constructed, the area of any rectangular region of the image can be computed with four table accesses in constant time as follows:

$$a(x_1, y_1, x_2, y_2) = sat(x_2, y_2) - sat(x_1, y_2) - sat(x_2, y_1) + sat(x_1, y_1)$$

where $x_1 \leq x_2, y_1 \leq y_2$

The SAT technique was introduced for texture mapping, but it was later applied to a wider class of problems including texture mapping [5], face detection [26], real time rendering of glossy environmental reflections [8], interactive glossy transparency rendering [12], empty space removal [17], and ambient

occlusion and halo generation [21]. A potential application disadvantage is that a SAT may require many bits per entry than may be available for a given data type. The number of bits required to store each element in SAT with a $w \cdot h$ resolution is $N = \log_2 w + \log_2 h + P_i$, where P_i is the precision required by the input [6]. This may complicate SAT handling for larger inputs [8]. Summed area tables may be built incrementally on a CPU with a cost that is linear with the number of cells in the original table.

3.6 Summed Volume Tables

Summed volume tables are a multidimensional extension [8] of SATs used in applications including rapid processing of volumetric information for rendering three-dimensional objects [6] and to process sequences of 2D frames as a volume [9]. The construction of a summed volume table is similar to construct of a SAT and it can also be done incrementally. A summed volume table, referred to as an integral volume in the following description for short, denoted as $iv(x,y,z)$ contains the sum of volumetric values for all the 3D positions between the origin and the (x,y,z) position, formally:

$$iv(x, y, z) = \sum_{\substack{x' \leq x \\ y' \leq y \\ z' \leq z}} i(x', y', z')$$

Where $i(x,y,z)$ is the input 3D information. The integral volume can be computed rapidly in one pass over the 3D input using the following equations:

$$s_1(x, y, z) = s_1(x, y - 1, z) + i(x, y, z)$$

$$s_2(x, y, z) = s_2(x - 1, y, z) + s_1(x, y, z)$$

$$iv(x, y, z) = iv(x, y, z - 1) + s_2(x, y, z)$$

where $s_1(x, -l, z)$, $s_2(-l, y, z)$, and $iv(x, y, -l)$ are all defined as zero.

In order to compute the volumetric properties of any rectangular prism aligned with the coordinate axes only eight array references to the integral volume are necessary [6]. For instance, the volume of the prism in Fig. 18 can be computed in constant time by the following eight references to the integral volume: $V_5 - V_1 - V_6 - V_7 + V_2 + V_3 + V_8 - V_4$.

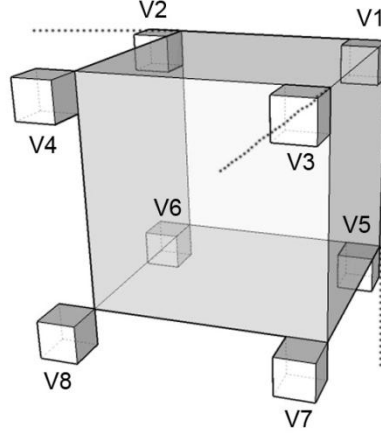


Figure 16: Calculating the sum of volumetric properties of a rectangular prism using a summed volume table

3.7 Velocity Perturbation Smoothing with Summed Volume Tables

As mentioned above, redundant computations can be eliminated and implicit parallelism of operations in the STA smoothers can be achieved using summed volume table techniques. For the cell smoother, the sums of cell velocity perturbations for all the smoothing volumes can be computed reading the value of each cell only once by calculating the integral volume of velocity perturbations. In a similar fashion, redundancy of coverage sums can be eliminated by computing the integral volume of coverage. Then, all the smoothing volumes for the seismic model can be computed in parallel by $n_x \cdot n_y \cdot n_z$ instances of a task that performs the same computations for each element of a model – an embarrassingly parallel process – with n_x elements in the X direction, n_y elements in the Y direction, and

n_z elements in the Z direction. For the vertex smoother, the algorithm constructs only one integral volume to store the sum of vertex velocity perturbations, as the number of cells in each smoothing volume can be directly calculated using the volume size. In a GPU implementation of the STA smoothers, the computation of each smoothing volume is performed by a CUDA kernel as described in the following chapters. Also presented there is a discussion of considerations for parallelizing the algorithms, execution times, detailed execution profiling, and use of the memory hierarchy of the parallel implementation of the smoothing algorithms.

Chapter 4: STA Smoothing Algorithms Analysis for Parallelization

4.1 Parallel Algorithm Design

The parallelization of the smoothing algorithms is based on Foster's parallel algorithm design method [7], which includes the following steps: partitioning, communication, agglomeration, and mapping. The goal of partitioning is to discover as much parallelism as possible without considering whether the discovered parallelism will survive other steps of the method to be reflected in the actual implementation. Partitioning divides the computations and data into their most primitive components either taking a computation centric approach or a data centric approach. For instance, the tasks of the STA cell smoother, which makes use of ray coverage, can be coarsely functionally partitioned as represented in Table 2. Similarly the vertex smoother can be partitioned as shown in Table3.

Table 2: Cell smoother partitioning

Cell Smoother Partitioning	Task Name
Read cell sums of velocity perturbations	READ DUSUM
Read cell coverage	READ COVER
Compute SVT of velocity perturbations	DUSUM SVT
Compute SVT of cell coverage	COVER SVT
Compute smoothed velocity perturbations	SMOOTH DUSUM/COVER

Table 3: Vertex smoother partitioning

Vertex Smoother Partitioning	Task Name
Read vertex velocity perturbations	READ DU
Compute SVT of velocity perturbations	DU SVT
Compute smoothed velocity perturbations	SMOOTH DU

To use some domain decomposition, computations to associate with each subset of partitioned data are determined with a special focus on the largest, most frequently accessed data structure. For a functional decomposition a dependence graph will indicate which tasks can be performed in parallel. For

the cell smoother, for instance, the computations of the SVT's require that the input files be read; and the computation of the smoothing volume requires that the SVT's have been computed as shown in Fig. 20.

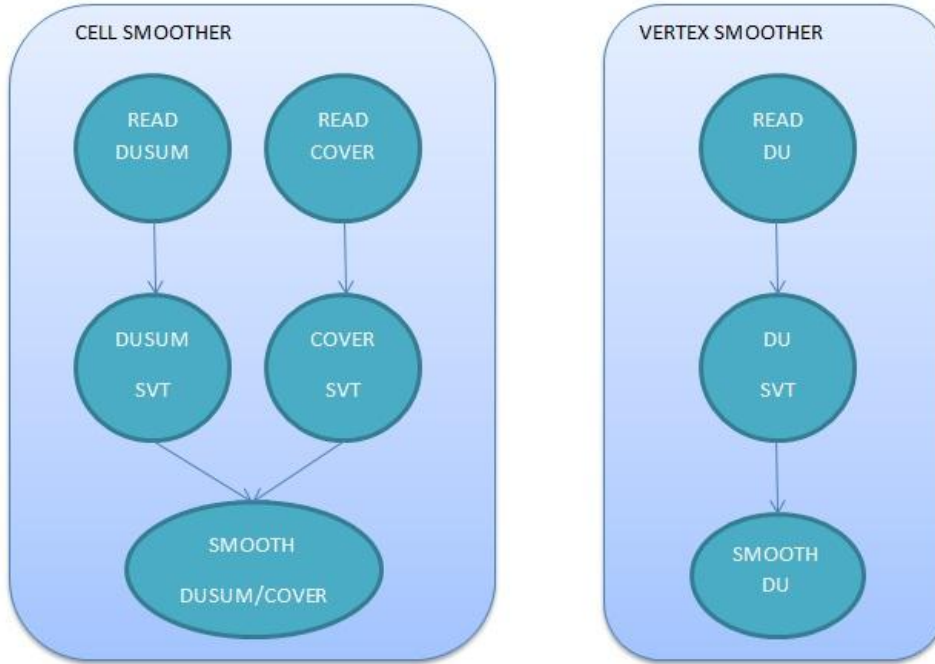


Figure 17: Data Dependence Graphs of STA Smoothers

The next step in Foster's method is to determine the communication pattern between tasks, which can be global or local. The former is used is used when a great number of tasks must contribute data to perform a computation. It is early in the method, but since it is not a strict waterfall process, it is worth mentioning that tasks mapped to the GPU can communicate globally through the device global memory and locally through share memory accessible to a warp. Communication is a major part of overhead of the parallel algorithm which may not be present in the sequential smoothing algorithms. Thus, minimizing parallel processing overhead is an important goal. One way to reduce overhead is to balance communication operations among tasks. Each task should communicate with only a small number of neighbors and perform communications and computations concurrently.

For both parallel smoothers, communication overhead is generated if SVTs are transferred between the CPU and the GPU – three relatively huge transfers that may eliminate the performance advantage introduced by parallel computations in the GPU.

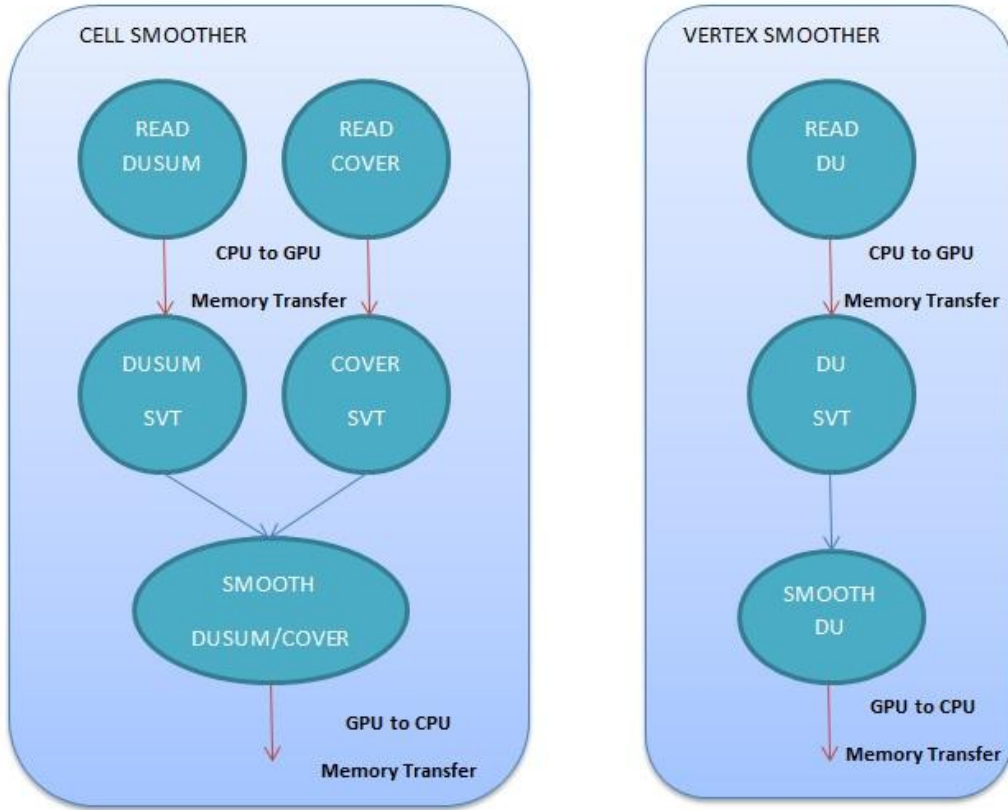


Figure 18: Potential data transfers between the CPU (host) and the GPU (device)

The third step of the parallel algorithm design method is agglomeration. The goal is to group tasks into larger tasks to improve performance, by lowering the communication overhead, or simplify programming, by removing parameter passing, result returning, and combining expressions of dependent computations separated during partitioning. Communication is a costly operation when developing GPU parallel code, as data needs to be copied from the host memory to the device and vice versa, but agglomeration can only happen amongst primitive tasks that will be executed on the same processor given the mapping target heterogeneous architecture.

Although, based on the domain decomposition of the cell smoother, both file readings and both SVT calculation steps can be computed in parallel, they are executed sequentially in the host to avoid communication and synchronization costs.

Finally, mapping assigns tasks to processors. On a centralized multiprocessor, as in multicore architectures, the operating system automatically maps processes to processors. The goals of mapping are to maximize processor utilization and minimize costly inter-processor communication. Processor utilization is the average percentage of time the processors are actively executing tasks.

At present, parallel code can be mapped on multicore CPUs or on GPUs. A powerful commercially available desktop computer can contain CPUs with several cores, as in dual- or quad-core processors and dual quad-core processors (eight cores). A very basic GPU may contain a stream multiprocessor with 16 cores. For instance, the GPUs of the Cyber-ShARE Center [5.2] used to run the parallelized STA smoothers contain 352 cores each and they can execute a maximum of 1536 concurrent threads. This significant advantage justified mapping the parallelized STA smoothers to a heterogeneous CPU/GPUs architecture. The next section presents a brief background on the GPU architecture and the CUDA programming model which used for this study.

4.2 GPU Architecture

Heterogeneous computers have a combination of central processing units (host) and graphical processing units (device). In a GPU-based application, the host executes most of the sequential section of the application, while the devices execute the parallel sections of the application divided in threads that increase the number of operations per second for the application. Sample specifications for a GPU device are shown in Fig. 22 and Table 4.

The standard Fermi streaming multiprocessor (SM) architecture [18] has at least eight stream processors (SPs) or cores, two special functional units (SFUs), a multithread instruction unit, an instruction cache, a global constant memory with 64kb, and a shared memory of 16kb. Each SP is a

hardware multithreading processor that can run more than 512 lightweight threads and has a scalar integer and floating-point arithmetic unit that executes most of the instructions. Each core has an RF and thread instructions that can use the SPs functions like sine, cosine, reciprocal, and square root at one result per cycle.

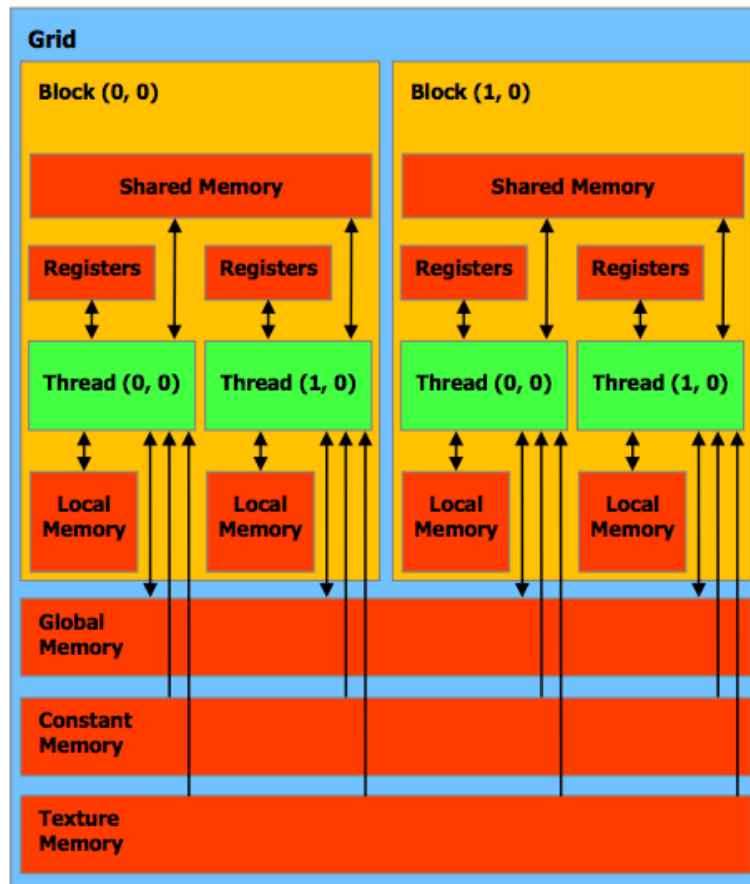


Figure : GPU memory structure [17]

The following section presents a brief background on the GPU architecture and programming model used for this study, which is based on the Nvidia Fermi architecture and the CUDA programming model.

Table 4: NVIDIA QUADRO 5000 GPU Device Specifications

NVIDIA QUADRO 5000 GPU Device	
CUDA Driver Version:	4.0
CUDA Capability Major/Minor version:	2.0
Total amount of global memory:	2559 MBytes (2683502592 bytes)
(11) Multiprocessors x (32) CUDA Cores/MP:	352 CUDA Cores
GPU Clock rate:	1.03 GHz
Memory Clock rate:	1500.00 Mhz
Memory Bus Width:	320-bit
L2 Cache Size:	655360 bytes
Max Texture Dimension Sizes	1D=(65536) 2D=(65536,65535) 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers	1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	32768
Warp size:	32
Maximum number of threads per block:	1024
Maximum sizes of each dimension of a block:	1024 x 1024 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 65535
Texture alignment:	512 bytes
Maximum memory pitch:	2147483647 bytes
Concurrent copy and execution:	Yes with 2 copy engine(s)
Run time limit on kernels:	Yes
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Concurrent kernel execution:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support enabled:	No
Device is using TCC driver mode:	No
Device supports Unified Addressing (UVA):	Yes
Device PCI Bus ID / PCI location ID:	4 / 0

The SMs have a single-instruction multi-thread (SIMT) architecture to manage and schedule 32 threads in parallel known, which are referred to as warps. The global memory, which is on off-chip DRAM, provides storage and communication for different thread blocks and SPs. Read/write shared

memory is visible for the block threads in each SP. Shared memory reduces access latency and provides a high bandwidth communication medium. Read constant memory is stored in DRAM.

The STA smoother implementation presented in this thesis uses more than 25,000 registers for local variables and global memory for the model arrays.

4.3 Summed Volume Tables for Parallel STA Smoothers

CUDA or Compute Unified Device Architecture is a parallel computing architecture developed by Nvidia. CUDA is the computing engine in Nvidia GPU's. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest Nvidia GPUs become accessible for computation like CPUs. Unlike CPUs however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. This approach of solving general purpose problems on GPUs is known as GPGPU. [17].

A CUDA-based algorithm can be used to improve the SVT construction times. Execution times are better for the CUDA version as the model size increases, since they are substantially smaller than the time required to load big volume models. However, the bottleneck of the process is the device data communication bus, as the results cannot remain in GPU global. For a GPU implementation the computation algorithm needs to change [6] to account for level size in the available memory hierarchy. For instance, table computations can be separated as follows:

$$\begin{aligned}
SX(i, j, k) &= \sum_{x < i} in(x, j, k) \\
SX(i, j, k) &= \sum_{y \leq j} SX(i, y, k) \\
SAT(i, j, k) &= SZ(i, j, k) \\
&= \sum_{z < k} SY(i, j, z)
\end{aligned}$$

$$\begin{aligned}
&= \sum_{z \leq k} \sum_{y \leq j} SX(i, y, z) \\
&= \sum_{z \leq k} \sum_{y \leq j} \sum_{x \leq i} in(x, y, z)
\end{aligned}$$

1. First pass SX computation in Z slices – load each line in shared memory, perform parallel computation of the line, and write it back.
2. Single-pass SY computation – read memory in a GPU coalesced read pattern.
3. Single-pass SZ computation – read memory in a GPU coalesced read pattern [6].

Notice the strong loop-carried data dependence due to the table creation order. This means that the maximum level of parallelism exists in plane creation of planes, with a maximum of three planes created in parallel (xy, xz and zy), but that is not the case for in-volume elements. However, calculation of in-volume elements can be based on recursive parallel plane calculations with decreasing plane sizes, such each set of planes is a level inside the planes just calculated. By parallelizing plane construction, SVT construction performance could peak at 2/3 of the sequential table creation time. Since that processing time is less than 5 milliseconds per table for a typical seismic model, the potential computational performance gain from this parallelization is under two milliseconds. However, the table transfer time to device memory reduces modest gain obtained from computations. Thus, the presented implementation constructs smoother SVTs on the host with a construction cost that is linear with the number of cells in the table [6].

Chapter 5: Results

This chapter presents the performance improvements obtained by removing redundancy and introducing parallel processing in the smoothing steps of Hole-Vidale's seismic tomography algorithm. Results presented include summed volume table generation times and the execution times of each of the smoothing methods.

5.1 Experimental Settings and Data Set

Amethyst is a collaborative visualization system also known as C2ViS. The main purpose of this visualization wall is to provide a high performance computing cluster service, and to display high-resolution images of scientific datasets for monitoring, exploratory, educational and outreach purposes. The C2ViS Tiled Display provides the ability to perform visualizations on a 9x5 tiled display of 40-inch NEC monitors resulting on a 93 Megapixel resolution allowing visualizations at a very high level of detail. Using 45 nodes to distribute workloads, this cluster has installed on each of its 45 nodes an Nvidia Quadro 5000 GPU that delivers 352 Cuda cores, along with an Intel Xeon with 8 cores (16 w/hyper-threading) and 12 GB of RAM [5.2].

Base Hardware included on all workstations by default which includes and is not limited to:

- Intel® 5520 chipset
- Intel® Xeon® Quad Core
- Nvidia Quadro 5000
- 12GB of RAM
- 1.5 TB SATA Hard drive
- Integrated Broadcom® 5754 Gigabit Ethernet controller (x2)

Extra features installed in this Windows I/O Channel CPU:

- Nvidia Quadro 5000 (x2)
- Infiniband ConnectX-2 40Gb/s NIC
- Intel PRO/1000 PF Server Adapter
- BlackMagic Design card

The Potrillo Volcanic Field (PVF) experiment, recorded in May 2003 is the test model for this investigation. Potrillo stretches through a 205 km long profile, consisting of 8 shots and 793 receivers across southern New Mexico and Far West Texas, and was designed as a detailed seismic investigation of the structure and composition of the Southern Rio Grande Rift and the Potrillo Volcanic field [1].

The execution of each algorithm was calculated through 36 iterations. Every six iterations, the cell and vertex smoothers were updated to reflect the smoothing schedule in Fig. 8.

5.2 Cell Smoother Performance Analysis

Execution times for the cell smoother, which utilizes an SVT for ray coverage in addition to the SVT for cell velocity perturbation sums, are greater than corresponding times for the vertex smoother. This translates into twice as many memory accesses for the former smoother when compared to the latter.

Fig. 23 shows execution times in milliseconds per iteration for each variation of the cell smoothing algorithm. The original sequential algorithm with a peak time of up to 113716 milliseconds is clearly outperformed by all other algorithms. That algorithm also shows a performance that is proportional to the size of the smoothing volume. As the smoothing volume gets smaller, the processing time decreases as well. The smoothers in this experiment change size every six iterations, always decreasing in size. This is caused by the large number of sequential and redundant operations needed for each smoothing volume. As the smoothing volume size decreases, fewer access and numerical operations are required and the volume traverses the model faster.

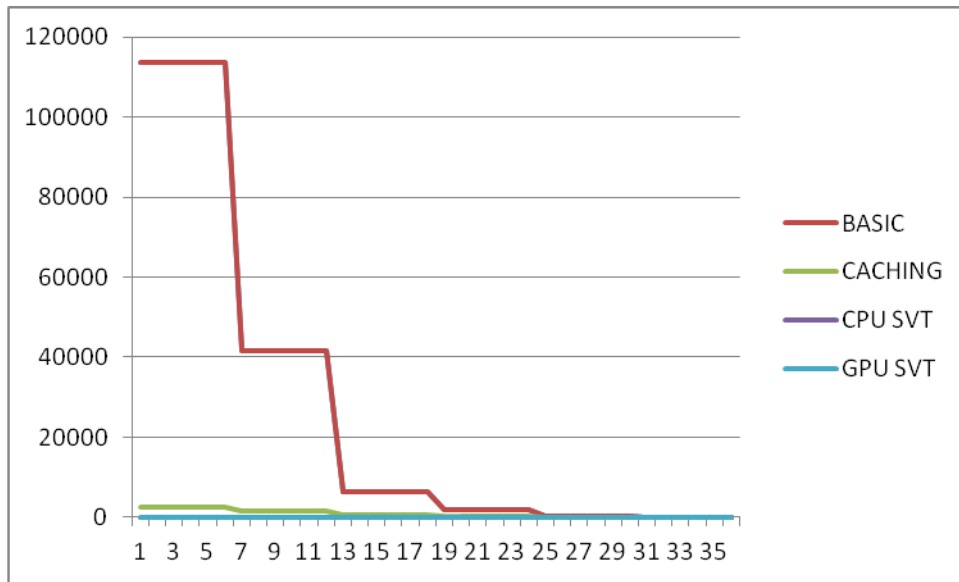


Figure 19: Execution time in milliseconds per iteration for the different versions of the cell smoother

With the implementation of the caching cell smoother, as shown in Fig. 23, the peak time was reduced to 2587 milliseconds worst case, which is the largest smoothing volume as in the basic algorithm. For both the basic and the caching algorithm, execution time decreases as the smoothing volume size decreases.

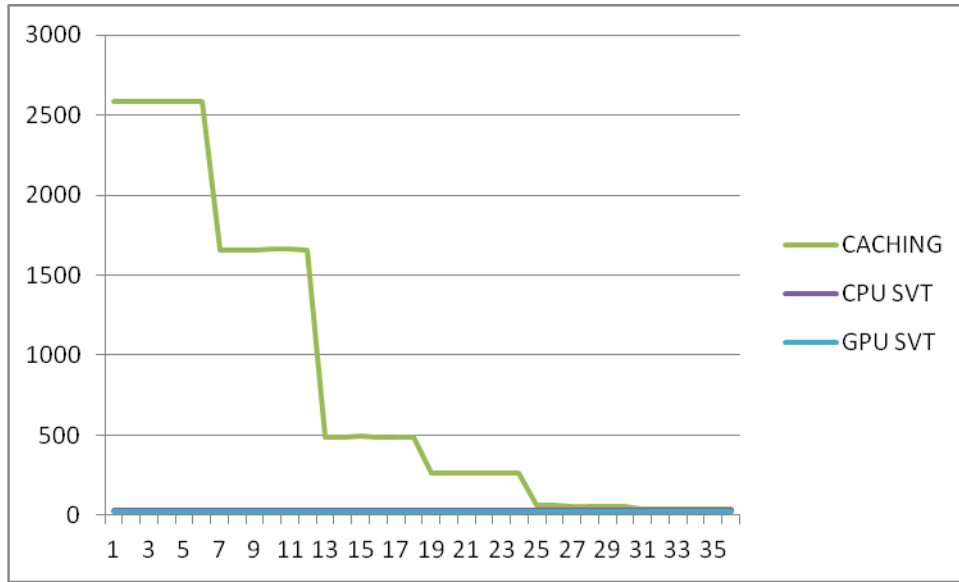


Figure 20: Execution times for the cell smoother with an emphasis on the caching technique, which is plotted in green

Finally, applying the SVT constant time smoothing algorithm, the parallel GPU-based approach can be compared with the CPU-based sequential approach. Even though executions times peak at 26 and 31 milliseconds, respectively, the parallelized algorithm has an 20.9% performance improvement with respect to the sequential smoother. In addition, the parallel version will show an even greater improvement when applied to larger models because of the available processing cores in the GPU. The larger the data, the more computations will be required by the smoother. For instance, since the GPU used to run the parallelized smoothers allows concurrent memory handling and code execution, the large models can be partitioned and transferred during the computation time.

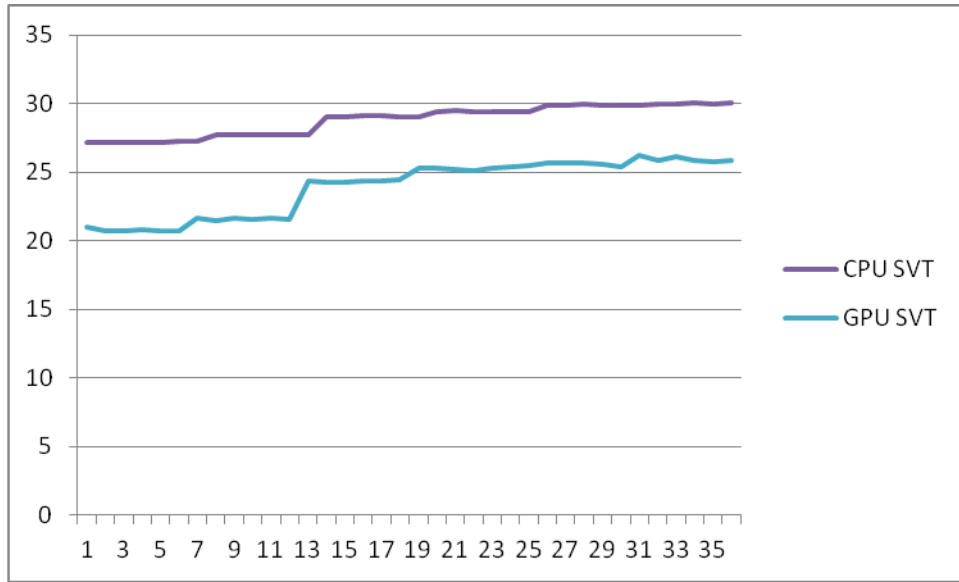


Figure 21: Execution time comparison for the parallel and sequential versions of the SVT cell smoothing algorithm

Comparing the best performance improvement with respect to the basic sequential algorithm, the caching cell smoother algorithm achieves an advantage of 44x speedup, the sequential SVT achieves a 4181x speedup, and the parallel SVT algorithm achieves a very impressive 5491x top speedup. The average performance improvement with respect to the basic sequential algorithm is 15.5x speedup, 994x speedup, and 1288x speedup for the caching, sequential SVT, and parallel SVT algorithms, respectively.

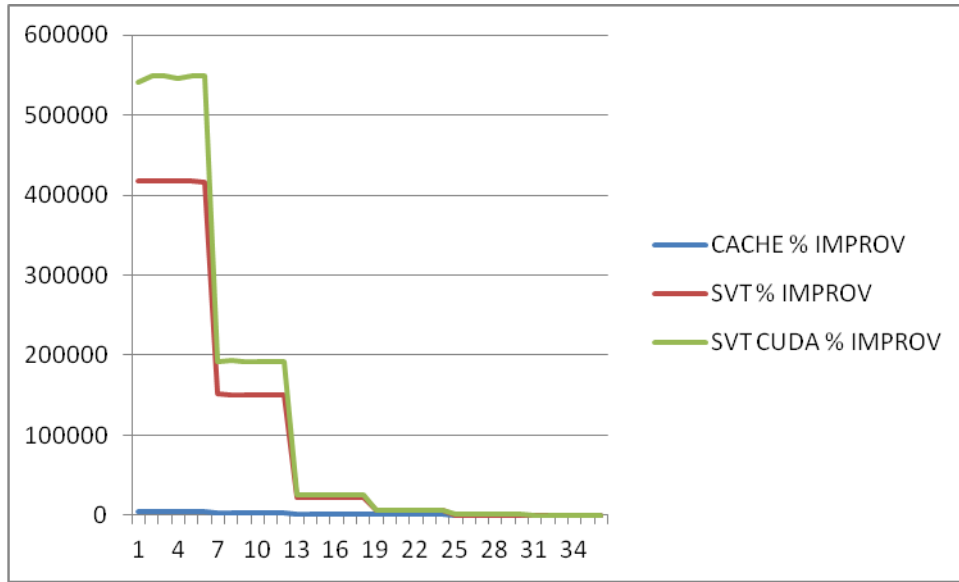


Figure 22: Percentage of improvement per iteration per algorithm compared with the basic cell smoother

Table 5: Execution times in milliseconds per iteration of the STA cell smoothing algorithms

CELL SMOOTHER						
BASIC	CACHING	SVT	SVT CUDA	CACHE % IMPROV	SVT % IMPROV	SVT CUDA % IMPROV
113716.1	2587.7	27.2	21	4294.485	417973.9	541405.2
113670.2	2584.5	27.2	20.7	4298.151	417805.1	549031.4
113658.6	2585.3	27.2	20.7	4296.341	417762.5	548975.4
113676.3	2584	27.2	20.8	4299.238	417827.6	546420.7
113656.7	2584.5	27.2	20.7	4297.628	417755.5	548966.2
113672.1	2584.1	27.3	20.7	4298.905	416281.3	549040.6
41541.9	1656.6	27.3	21.7	2407.66	152068.1	191337.3
41529.3	1657.3	27.7	21.5	2405.841	149825.3	193059.5
41544.1	1657.4	27.7	21.7	2406.583	149878.7	191347.5
41532.5	1661.4	27.7	21.6	2399.85	149836.8	192180.1
41524.1	1661.6	27.7	21.7	2399.043	149806.5	191255.3
41537.2	1657.4	27.7	21.6	2406.166	149853.8	192201.9
6332.7	487.2	27.7	24.4	1199.815	22761.73	25853.69
6332.7	487.5	29	24.3	1199.015	21736.9	25960.49
6332.4	491.6	29	24.3	1188.12	21735.86	25959.26
6332.7	487.7	29.1	24.4	1198.483	21661.86	25853.69
6334.4	489.2	29.1	24.4	1194.849	21667.7	25860.66
6332.5	487.3	29	24.5	1199.507	21736.21	25746.94

1718	264.8	29	25.3	548.7915	5824.138	6690.514
1719.1	261.6	29.4	25.3	557.1483	5747.279	6694.862
1718.2	261	29.5	25.2	558.3142	5724.407	6718.254
1718.8	261.4	29.4	25.1	557.5363	5746.259	6747.809
1718.6	260.9	29.4	25.3	558.7198	5745.578	6692.885
1718.3	260.9	29.4	25.4	558.6048	5744.558	6664.961
170.9	60	29.4	25.5	184.8333	481.2925	570.1961
170.8	58.7	29.9	25.7	190.971	471.2375	564.5914
170.3	58.3	29.9	25.7	192.1098	469.5652	562.6459
170.1	58.3	30	25.7	191.7667	467	561.8677
170.7	58.3	29.9	25.6	192.7959	470.903	566.7969
170.4	58.4	29.9	25.4	191.7808	469.8997	570.8661
64.2	35.3	29.9	26.2	81.86969	114.7157	145.0382
64.4	35.3	30	25.9	82.43626	114.6667	148.6486
64.3	35.3	30	26.1	82.15297	114.3333	146.3602
64.2	35.3	30.1	25.9	81.86969	113.289	147.8764
64.2	35.3	30	25.8	81.86969	114	148.8372
64.2	35.3	30.1	25.9	81.86969	113.289	147.8764

5.3 GPU Performance Analysis of STA Cell Smoothing Algorithm

The bottleneck in GPU computing is usually the memory transfer rate. Input data for computations in the GPU have to be transferred from the host memory to the device memory and vice versa. For large data transfers, worst-case communication time can be greater than overall sequential computation time. For the presented parallelized cell smoothing algorithm, data transfers to the device are minimized as much as possible to reduce the transfer time overhead. The following plots and tables show time percentages computed by the CUDA profiler [17]. GPU calculations took approximately 75.3% of time, host to device memory transfers took 16.8% of time, and device to host memory transfers took 2.8% of time.

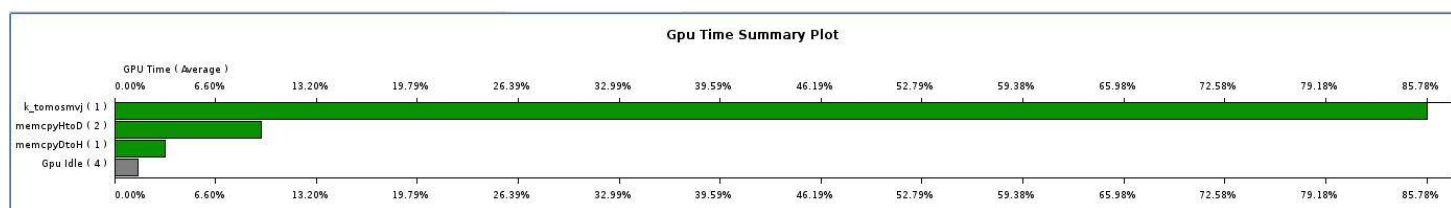


Figure 23: GPU time summary plot for the cell smoother.

Table 6: GPU time breakdown for the cell smoother

	Method	#Calls	GPU time (us)	CPU time (us)	%GPU time
1	k_tomosmvj	1	7167.84	7180.84	75.33
2	memcpyHtoD	2	1595.52	1981	16.77
3	memcpyDtoH	1	270.752	8878	2.84

Table 7: GPU time breakdown for kernels, memory transfers and occupancy for the cell smoother

	GPU Timestamp (us)	Method	GPU Time (us)	CPU Time (us)	grid size	thread block size	registers per thread	Occupancy
1	0	memcpyHtoD	912.512	1143				
2	1290	memcpyHtoD	683.008	838				
3	2072	k_tomosmvj	7167.84	7180.84	[3 2 1]	[32 16 1]	49	0.333
4	9244	memcpyDtoH	270.752	8878				

The width plot below shows any concurrent operations. Memory transfers from host to device show a small offset in memory readings during initialization due to file accesses performed by the CPU to gather data and add it to the array which is later transferred. This transfer time can be eliminated by combining the cell and vertex smoothers into a single process. In this case, the host would take and pass the output from the cell smoother to the vertex smoother without taking time to save and read cell smoother results from a file. However, this is out of the scope of this thesis and it would also cause the loss of the intermediate files which can be used for processing assessment through analysis and visualizations.

The next figure shows CPU execution time, which is concurrent with GPU execution, as the host has to invoke the kernels to be computed in the device. In addition, the host performs several other calculations after smoothing finishes, including file accesses.

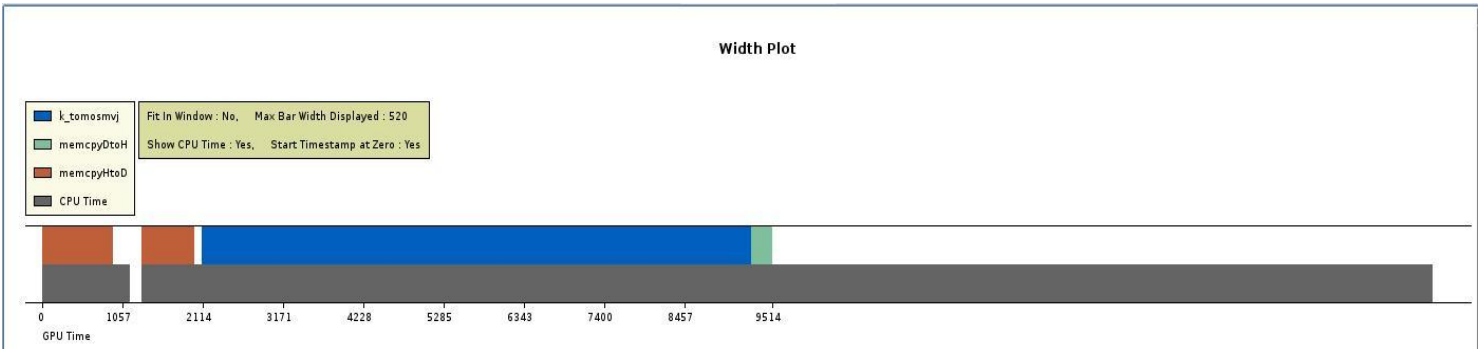


Figure 24: With plot with overlapping CPU and GPU times for the cell smoother.

The following table shows occupancy, which is the number of kernel calls times the average processing time of each kernel over the number of threads available. The higher the occupancy, the better the GPU use will be. There are several factors that limit occupancy. In this case, 512 threads are concurrently launched by the cell smoothing algorithm. Although the device can have a maximum of 1536 active threads, memory registers are a limiting factor to achieve a greater occupancy.

Table 8: Occupancy analysis for cell smoothing kernel

Occupancy analysis on device Quadro 5000	
Grid size	[3 2 1]
Block size	[32 16 1]
Register Ratio	0.78125 (25600 / 32768) [49 registers per thread]
Shared Memory Ratio	0 (0 / 49152) [0 bytes per Block]
Active Blocks per SM	1 (Maximum Active Blocks per SM: 8)
Active threads per SM	512 (Maximum Active threads per SM: 1536)
Potential Occupancy	0.333333 (16 / 48)
Occupancy limiting factor	Registers

Device memory management, unlike the host's, requires manual handling of each level, namely, global memory, shared memory, texture memory, and registers. For the cell smoother implementation, code execution and concurrent memory handling are limited by the following two factors:

1. Large data transfers to device memory are needed by device computations to start, as initial smoothing volumes sizes are a large percentage of the model size. Since smoothing is a fast operation in the GPU, the overlap of communication and computation is minimal and SP occupancy is low.

2. The properties of the device memory hierarchy. Global memory, which is the largest and also the slowest, is used by the cell smoother to transfer two 1.6MB arrays, each with 500,000 elements. Access to device shared memory, which is faster than global memory but also read by all kernels in a warp, would require use of synchronization locks and would reduce kernel concurrency. Another problem with shared memory is that only 48KB are available per block, which is smaller than the space required by the cell smoother and would require explicit transfer of sub-blocks up and down the device memory hierarchy. Similarly, although the texture memory contains its own cache, it has also a limited, 64KB of space and it is read only for kernels, which means that the smoother results cannot be stored in texture memory. Finally, registers, which is the fastest device memory level, are only 32k registers and they are used to store kernel local variables. Thus, kernels are limited to mainly use device global memory. In addition, since the model sizes is relatively small for the global memory, its bandwidth utilization is low.

The performance impact of the limitations above could be lessened, but the necessary the effort would be high compared to the potential benefit. For instance, optimizations considering the limitations listed above would apply to a very limited variety of GPU models and would reduce portability of the algorithm implementation. In addition, the total time spent on memory input and output is approximately 20% of all GPU time, which translates into a mere 6 milliseconds in the worst case, and since memory transfers are still required, the maximum potential time saved would be less than 6 milliseconds.

Table 9: Memory throughput analysis for cell smoothing kernel on device Quadro 5000

Memory throughput analysis for cell smoothing kernel on device Quadro 5000	
Kernel requested global memory read throughput(GB/s)	2.25
Kernel requested global memory write throughput(GB/s)	0.23
Kernel requested global memory throughput(GB/s)	2.49
L1 cache read throughput(GB/s)	60.03
L1 cache global hit ratio (%)	1.61
Texture cache memory throughput(GB/s)	0.00
Texture cache hit rate(%)	0.00
L2 cache texture memory read throughput(GB/s)	0.00
L2 cache global memory read throughput(GB/s)	57.84
L2 cache global memory write throughput(GB/s)	1.86
L2 cache global memory throughput(GB/s)	59.70
Local memory bus traffic(%)	0.00
Global memory excess load(%)	96.10
Global memory excess store(%)	87.50
Achieved global memory read throughput(GB/s)	24.82
Achieved global memory write throughput(GB/s)	1.86
Achieved global memory throughput(GB/s)	26.68
Peak global memory throughput(GB/s)	120.00

To complement data used for the analysis of the cell smoothing algorithm, a summary table is annexed with the most basic information about the kernel.

Table 10: Analysis for cell smoothing kernel on device Quadro 5000.

Summary profiling information for the kernel	
Number of calls	1

GPU time(us)	7167.84
GPU time (%)	75.33
Grid size	[3 2 1]
Block size	[32 16 1]

5.4 Performance Analysis of STA Vertex Algorithm

The analysis of the STA vertex smoothing algorithm follows very closely the analysis presented above for the cell smoother. Since the same memory restrictions apply for both smoothing algorithms, only the results of the vertex smoother are discussed along with a few key differences.

Execution times of the vertex smoother are faster than corresponding times of the cell smoother, as coverage information is not needed for smoothing vertex velocity perturbations. Again plots and tables show execution times in milliseconds per iteration for each variation of the vertex smoothing algorithm. With a peak time of 97731 milliseconds, the basic sequential algorithm has the worst performance. Also, the performance of the vertex smoothing algorithm is proportional to the smoothing volume size. Since the smoothing schedule is approximately the same for both smoothers, the size of the smoothing volume decreases every six iterations.

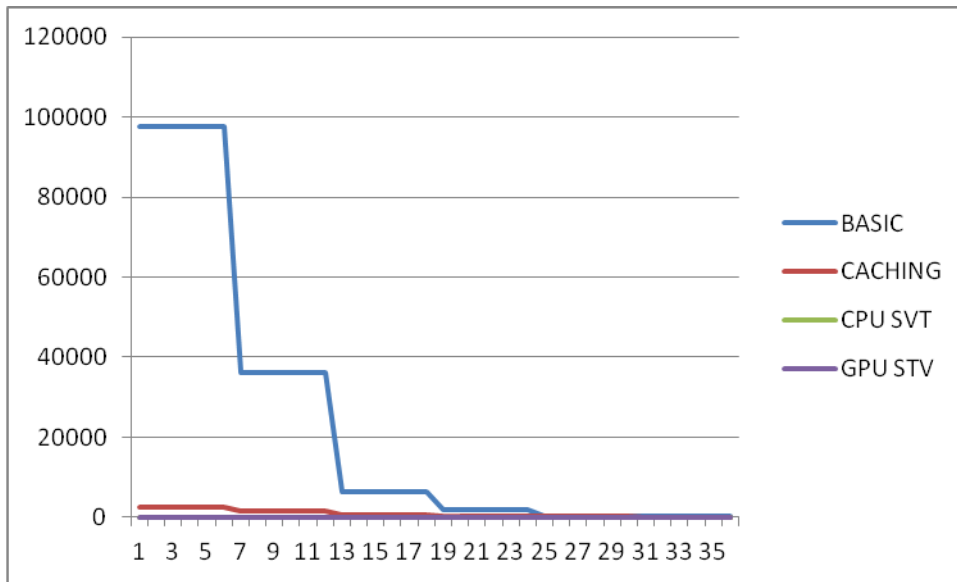


Figure 25: Execution time in milliseconds per iteration for the different versions of the vertex smoother

For the CPU implementation of the caching vertex smoother, the peak time was reduced to 2510 milliseconds on its worst case, similar to the cell smoothing algorithm peak of 2587 milliseconds on the first iteration.

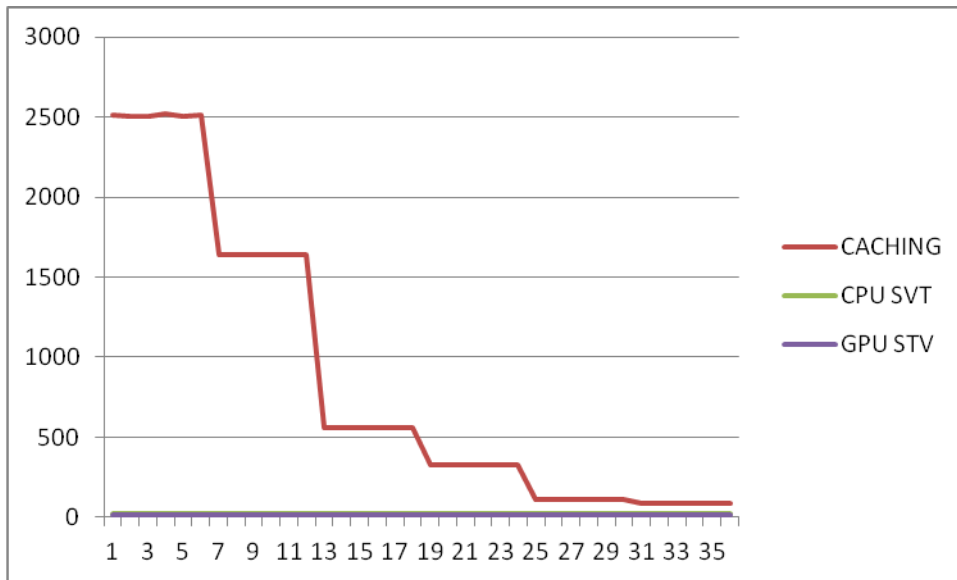


Figure 26: Execution times for the vertex smoother with an emphasis on the caching technique, which is plotted in maroon

For the SVT constant time vertex smoothing algorithm, comparison of the parallel GPU-based approach vs. sequential CPU-based approach shows an average 66.4% performance improvement – much better than the performance gains obtained with the SVT cell smoothing algorithms. Regarding smoother performance relationship to data to process, the larger the data, the better the benefit, as in the cell vertex smoothing case.

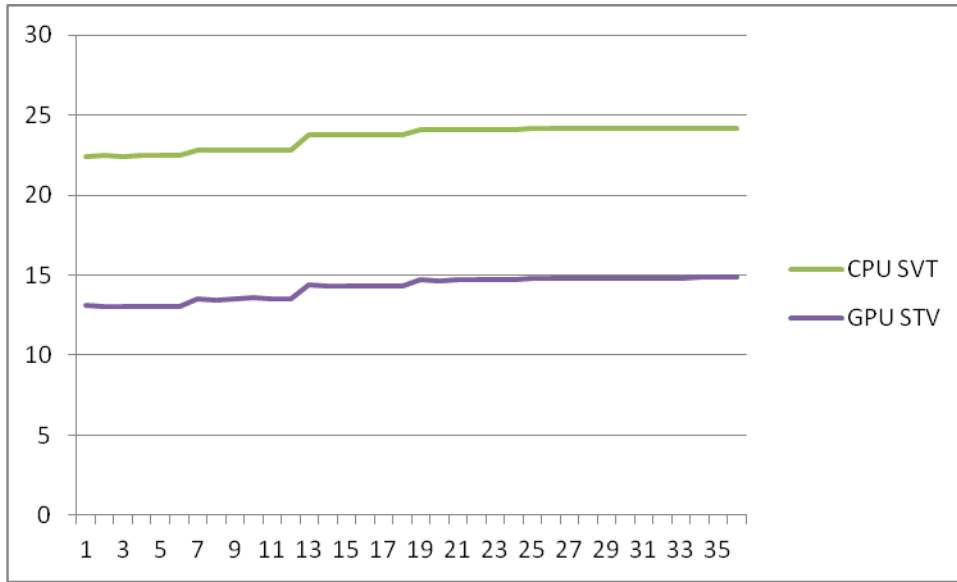


Figure 27: Execution time comparison for the parallel and sequential versions of the SVT vertex smoothing algorithm

For performance improvement vs. the sequential basic algorithm, caching algorithm achieves a 37.9x speedup, the sequential CPU-based SVT vertex smoother achieves a best-case 4362x speedup and the parallel GPU-based version achieves a best-case 7459.4% of speedup.

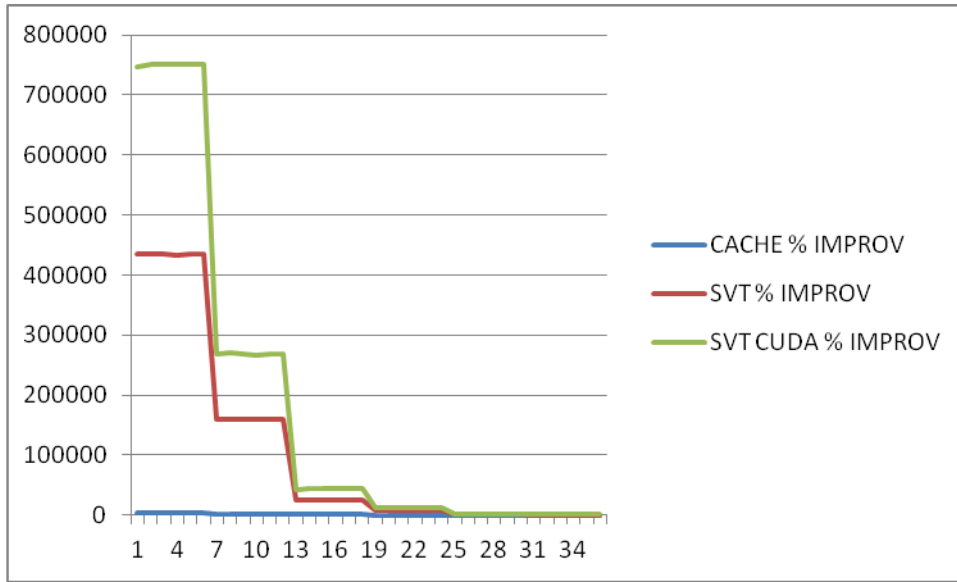


Figure 28: Percentage of improvement per iteration per algorithm compared with the basic vertex smoother

Table 11: Performance of the vertex smoothing algorithms

VERTEX SMOOTHER						
BASIC	CACHING	SVT	SVT CUDA	CACHE % IMPROV	SVT % IMPROV	SVT CUDA % IMPROV
97731.2	2510	22.4	13.1	3793.673	436200	745939.7
97703	2507.6	22.5	13	3796.275	434135.6	751461.5
97697.6	2508.1	22.4	13	3795.283	436050	751420
97672.5	2518.7	22.5	13	3777.893	434000	751226.9
97706.9	2506.5	22.5	13	3798.141	434152.9	751491.5
97707	2510.3	22.5	13	3792.244	434153.3	751492.3
36235.1	1643.7	22.8	13.5	2104.484	158825.9	268308.1
36253.7	1643.6	22.8	13.4	2105.75	158907.5	270450
36245.6	1643.3	22.8	13.5	2105.659	158871.9	268385.9
36269.2	1641.2	22.8	13.6	2109.92	158975.4	266585.3
36233.1	1640.2	22.8	13.5	2109.066	158817.1	268293.3
36218	1640.9	22.8	13.5	2107.203	158750.9	268181.5
6240.8	559.4	23.8	14.4	1015.624	26121.85	43238.89
6240.8	558.4	23.8	14.3	1017.622	26121.85	43541.96
6240.7	559	23.8	14.3	1016.404	26121.43	43541.26
6240.5	559.3	23.8	14.3	1015.77	26120.59	43539.86
6240.1	558.1	23.8	14.3	1018.097	26118.91	43537.06

6240.3	558.7	23.8	14.3	1016.932	26119.75	43538.46
1914.5	325.6	24.1	14.7	487.9914	7843.983	12923.81
1914.4	326.6	24.1	14.6	486.1604	7843.568	13012.33
1914.8	325.5	24.1	14.7	488.2642	7845.228	12925.85
1914.5	325.5	24.1	14.7	488.172	7843.983	12923.81
1914.5	325.4	24.1	14.7	488.3528	7843.983	12923.81
1916.8	325.5	24.1	14.7	488.8786	7853.527	12939.46
292.9	113.7	24.2	14.8	157.6077	1110.331	1879.054
292.8	113.7	24.2	14.8	157.5198	1109.917	1878.378
292.9	113.7	24.2	14.8	157.6077	1110.331	1879.054
292.9	113.7	24.2	14.8	157.6077	1110.331	1879.054
292.9	113.7	24.2	14.8	157.6077	1110.331	1879.054
293	113.7	24.2	14.8	157.6957	1110.744	1879.73
153	84.6	24.2	14.8	80.85106	532.2314	933.7838
153.1	84.6	24.2	14.8	80.96927	532.6446	934.4595
153.4	84.6	24.2	14.8	81.32388	533.8843	936.4865
153.3	84.5	24.2	14.9	81.42012	533.4711	928.8591
152.9	84.6	24.2	14.9	80.73286	531.8182	926.1745
153.2	84.6	24.2	14.9	81.08747	533.0579	928.1879

5.5 GPU Performance Analysis of the STA Vertex Smoothing Algorithm

As in the case of the cell smoother, host to device memory transfers are reduced to minimize the negative performance impact transfer times. The following figures and tables show the GPU performance data obtained from the CUDA profiler while the GPU performed calculations and memory transfers. Almost 83% was computing time, 9% host to device memory transfers, and 6% device to host memory transfers.

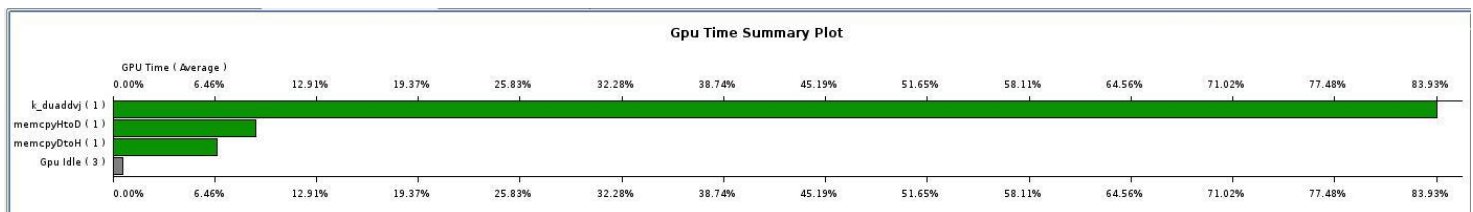


Figure 29: GPU time summary plot for the vertex smoother.

Table 12: GPU time breakdown for the vertex smoother

	Method	#Calls	GPU time (us)	CPU time (us)	%GPU time
1	k_duaddvj	1	3398.53	3410.53	82.99
2	memcpyHtoD	1	363.936	624	8.88
3	memcpyDtoH	1	263.776	4058	6.44

Table 13: GPU time breakdown for kernels, memory transfers and occupancy for the vertex smoother

	GPU Timestamp (us)	Method	GPU Time (us)	CPU Time (us)	grid size	thread block size	registers per thread	Occupancy
1	0	memcpyHtoD	363.936	624				
2	430	k_duaddvj	3398.53	3410.53	[3 2 1]	[32 16 1]	34	0.333
3	3832	memcpyDtoH	263.776	4058				

The figure below shows any concurrent operations. The first bar are data transfers from host to device. Again, there is a small offset in the memory readings during the initialization due to file accesses while the host CPU gathers information and adds information to the array which is later transferred. As in the cell smoother case, the negative impact of transfer overhead can be avoided by combining both smoothers into a single instance. The host would have to temporarily store the result of the cell smoother to use it as input for the vertex smoother. Next is the CPU time, which executes while the GPU executes kernels, invoked by the CPU. In addition, the CPU performs several other calculations after the vertex smoother finishes, including reading and writing to other files.

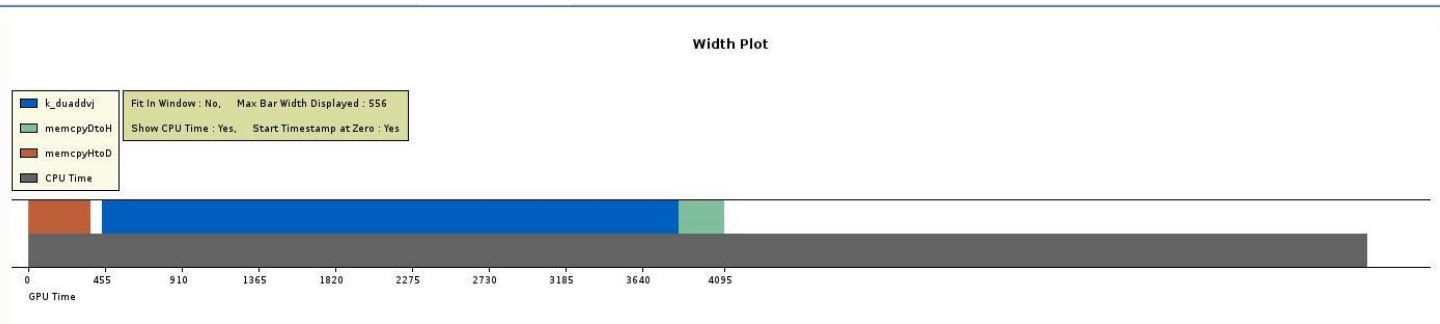


Figure 30: With plot with overlapping CPU and GPU times for the vertex smoother

The following table shows occupancy results for the vertex smoother. For this smoother, 512 threads are launched because of limitations in the number of available registers. Therefore, occupancy is identical to that of the cell smoother.

Table 14: Occupancy analysis for the vertex smoothing kernel

Occupancy analysis on device Quadro 5000	
Kernel details: Grid size	[3 2 1], Block size: [32 16 1]
Register Ratio	0.53125 (17408 / 32768) [34 registers per thread]
Shared Memory Ratio	0 (0 / 49152) [0 bytes per Block]
Active Blocks per SM	1 (Maximum Active Blocks per SM: 8)
Active threads per SM	512 (Maximum Active threads per SM: 1536)
Potential Occupancy	0.333333 (16 / 48)
Occupancy limiting factor	Registers

Some of the same memory limitations that affect the cell smoother also apply to the vertex smoother. However, there are two key differences. On the one hand, there are fewer memory transfers because ray coverage file is not needed. With memory transfers reduced to half, the memory transition time is reduced as well to about half of the corresponding time of the cell smoother, that is, about 10 milliseconds. This makes even less profitable any attempt to perform concurrent data transfers and computations. Total time spent in memory transfers is approximately 10% to 15% of all GPU time.

Since this algorithm executes faster, this percentage of time translates into 3 to 4 milliseconds approximately in the worst case. Since memory transfers are required at some level, the maximum potential time to save would be less than 4 milliseconds.

Table 15: Memory throughput analysis for vertex smoothing kernel on device Quadro 5000

Memory throughput analysis for vertex smoothing kernel on device Quadro 5000	
Kernel requested global memory read throughput(GB/s)	2.35
Kernel requested global memory write throughput(GB/s)	0.48
Kernel requested global memory throughput(GB/s)	2.83
L1 cache read throughput(GB/s)	60.66
L1 cache global hit ratio (%)	2.76
Texture cache memory throughput(GB/s)	0.00
Texture cache hit rate(%)	0.00
L2 cache texture memory read throughput(GB/s)	0.00
L2 cache global memory read throughput(GB/s)	56.75
L2 cache global memory write throughput(GB/s)	3.86
L2 cache global memory throughput(GB/s)	60.62
Local memory bus traffic(%)	0.00
Global memory excess load(%)	95.86
Global memory excess store(%)	87.50
Achieved global memory read throughput(GB/s)	5.81
Achieved global memory write throughput(GB/s)	3.86
Achieved global memory throughput(GB/s)	9.67
Peak global memory throughput(GB/s)	120.00

To complement the analysis of the vertex smoothing algorithm, a summary table is annexed with the basic kernel information. One important aspect to note is that although this algorithm requires only one file, not two files as the cell smoother, the overall benefit of memory transfer throughput depends on model size, not on the number of files. In other words, although fewer files should mean faster computation, memory throughput and concurrent execution of memory transfers and computing are affected only by the model size.

Table 16: Analysis for the vertex smoothing kernel on device Quadro 5000

Summary profiling information for the kernel	
Number of calls	1
GPU time(us)	3398.53
GPU time (%)	82.99
Grid size	[3 2 1]
Block size	[32 16 1]

Chapter 6: Conclusion

This thesis focused on the achievement of the following goals:

1. To perform a parallelization analysis of current sequential smoothing algorithms
2. To design, map, and implement parallelizable operations of such algorithms

The first goal was achieved by analyzing the three variations of the two types of smoothing algorithm of Hole-Vidale's STA. The second goal was attained by following Foster's method [7] to analyze, partition, agglomerate, and map algorithmic tasks. Volume processing extensions of the SAT algorithms are adapted in this thesis for geophysical structure reconstruction. Algorithmic tasks are mapped to a heterogeneous CPU/GPU architecture. Parallel tasks are implemented with the CUDA programming environment [17] and executed in an NVIDIA Fermi GPU [18]. Using a controlled-source seismic tomography data set, a parallel implementation of the STA cell smoothing algorithm on an NVIDIA Fermi GPU delivers a peak performance improvement of 31.9% and an average 20.9% improvement with respect to the fastest sequential counterpart of the algorithm. Similarly, the parallel implementation of the STA vertex smoother algorithm obtains a peak performance improvement of 73.1% and an average improvement of 66.4% with respect to the fastest sequential implementation of the algorithm. Since these improvements are a function of model and smoothing volume sizes, the parallel implementation of both the cell and the vertex smoother will outperform sequential implementations for larger models and smoothing volumes. In addition, the parallel implementations include the scalability benefit of using hundreds of stream processing cores in GPUs, which are expected to continue increasing in number of cores per device as processor architecture and manufacturing techniques continue to improve.

6.1 Future Work

Further smoother performance optimizations can be explored by fine-tuning of task memory management and access. Although the smoothing kernels have poor spatial and temporal locality when processing large smoothing volumes, performance could be increased by sub-block processing tuned for device cache memories. This could especially impact the smallest smoothing volumes. However, the small smoothing volumes are fast to process already and they may represent just 1/6 of the total smoothing schedule. In addition, tuning of memory access and management will require a significant effort for the aggregated, relatively small potential improvements. Unless larger models and more complex smoothing schedules are presented, memory handling modifications are unlikely.

The GPU SVT algorithm is designed specifically for the Amethyst cluster. Portability of this algorithm to other clusters and GPU devices, including enabling multiple GPU computing for enhance scalability of the STA algorithms are likely future research.

The increased performance of the GPU STV algorithm will enable several features previously used at the user's discretion due to long computation times. One technique to explore is adaptive smoothing with genetic algorithms. This technique may allow dynamic adjustment of smoothing schedule to accelerate model convergence. The GPU SVT can also enhance model fusion and cross-validation methods, as the needed multiple different converging models could be computed in parallel [2]. Obtained models would then be fused to produce a better image than a single solution obtained using the entire dataset at once [14] [19] [22]. Finally, one important goal of three-dimensional seismic tomography is to visualize the Earth's crust velocity structures for domain expert analysis. Visualization is an efficient way to rapidly analyze hundredths of thousands of numerical values that are binary coded in each module output, and allow scientists and researchers to discover artifacts, anomalies, or defective equipment in a fraction of the time it would take to analyze the binary output numerically. Faster smoothing takes one step further toward real time interaction and visualization of STA's output models.

References

- [1] Averill, M. 2007. “A Lithospheric Investigation of the Southern Rio Grande: Ph.D. Dissertation”, The University of Texas at El Paso.
- [2] Averill, M. Miller, K. Kreinovich, V. and Velasco, A. 2008. “Viability of Travel-Time Sensitivity Testing for Estimating Uncertainty of Tomography Velocity Models”, In Geophysics.
- [3] Balanis, C. and Bentley, D. 1986. “Algorithm and Filter Selection in Geophysical Tomography” IEEE Transactions, Geosciences, Remote Sensing, vol. GE-24 no. 6. Pp. 983 – 996 July.
- [4] Bentley, D. 1985. “Algorithm and Filter Selection in Geophysical Tomography: M.S.E.E. Thesis”, Arizona State University.
- [5] Crow, F. 1984. “Summed-Area Tables for Texture Mapping”, In Proceedings of SIGGRAPH 84, pp. 207 – 212.
- [5.2] Cyber-Share’s Amethyst Collaborative Visualization System (C2ViS), Amethyst Documentation, Technical Manual, 2011.
- [6] Díaz, J., Vázquez, P., et.al. “Real-Time Ambient Occlusion and Halos with Summed Area Tables, Computers & Graphics 34, 2010, pp. 337 – 350.
- [7] Foster, Ian. “Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering”, Reading, MA: Addison Wesley, 1995.
- [8] Glassner, A. 1990. “Multi-Dimensional Sum Tables”, In Graphics Gems pp. 376 – 381. Academic Press.
- [9] Hensley, J. Scheuermann, T., Coombe, G., Singh, M., and Lastra, A., 2005. “Fast Summed Area Table Generation and its Applications”, In Eurographics.
- *[10] Herman, G. and Leant, A. 1976. “Iterative Reconstruction Algorithms”, Computers, Biology and Medicine, vol. 6, pp. 273 – 294.
- [11] Hole, J. 1992. “Nonlinear High-Resolution Three-Dimensional Seismic Travel Time Tomography”, Journal of Geophysics Research pp. 6543 – 6552.
- *[12] Huang, T., Yang, G., and Yang, G., “A Fast Two-Dimensional Median Filtering Algorithm”, IEEE Transactions on Acoustics, Speech and Signal Processing, vol.27, pp. 13 – 18.
- [13] Kosloff, T., Tao, M., and Barsky, B. 2007. “Depth of Field Post Processing for Layered Scenes Using Constant – Time Rectangle Spreading”, University of California, Berkley.
- [14] Kreinovich, V., Miller, K., et.al., “Spatial Resolution for Processing Seismic Data: Type-2 Methods for Finding the Relevant Granular Structure”, The University of Texas at El Paso.

- *[15] Kwei, L. 1984. “Image Enhancement of Geophysical Tomographs: M.S.E.E. Thesis”, West Virginia University, Morgantown.
- *[16] Lytle, R. and Dines, A. 1980. “Iterative Ray-Tracing between Boreholes for Underground Image Reconstruction”, IEEE Transactions, Geosciences, Remote Sensing, vol. GE-18 no. 3, pp. 234 – 240 July.
- [17] NVIDIA CUDA Programming Guide 4.0, http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf
- [18] NVIDIA’s Next Generation CUDA Compute Architecture: Fermi, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
- [19.1] Olaya, J., Chacon, C., Velasco, A., and Romero, R., 2011. “Accelerating Velocity Perturbation Smoothing”, Innovative Parallel Computing INPAR 2012 (submitted), 2012
- [19] Olaya, J., 2010. “Fusion of Monte Carlo Seismic Tomography Models”, University of Texas at El Paso.
- [20] Papageorgiou, C., Oren M. and Poggio, T. 1998. “A General Framework for Object Detection”, In International Conference on Computer Vision.
- *[21] Perrault, S., and Hebert, P., 2007. “Median Filtering in Constant Time”, IEEE Transactions on Image Processing, pp. 2389 – 2394.
- [22] Peterson, J.E. Jr., and Davey A., 1991. “Crossvalidation method for Crosswell Seismic Tomography”, Geophysics Vol. 56 No. 3, March, pp. 385 – 389.
- [23] Romero, R., and Fuentes O., “Improving Smoothing Performance with Summed Volume Tables”, in preparation
- [24] Vidal, V., Mei, X. and Decaudin, P. 2006. “Simple Empty – Space Removal for Interactive Volume Rendering”, Journal of Graphics Tools, vol. 13, no. 2, pp. 21 – 37.
- [25] Vidale, J. 1990. “Finite-Difference Calculations of Travel Times in Three Dimensions”, Geophysics, pp. 521 – 526.
- [26] Viola, P., and Jones, M. 2004. “Robust Real-Time Face Detection”, International Journal of Computer Vision, pp. 137 – 154.
- [27] Zhang H., and Thurber, C., 2006. “Development and Applications of Double-Difference Seismic Tomography”, Pure and Applied Geophysics, 2006, Vol. 163, pp. 373 - 403

Vita

Ivan Gris Sepulveda

Education

B.S., Computer Science, 2010, University of Texas at El Paso, El Paso, Texas

M.S., Computer Science, 2011 (expected), University of Texas at El Paso, El Paso, Texas

Association Memberships

Member, Institute of Electrical and Electronics Engineers, Computer Society (IEEE CS)

Member, International Game Developers Association (IGDA);

Member, ACM Special Interest Group in Graphics (SIGGRAPH);

Member, Phi Kappa Phi Honors Society

Professional Experience

Cyber-Share Center of Excellence, University of Texas at El Paso

El Paso, TX

Dec08 – Present

Research Assistant

Designed and developed tools for the Geology department to display seismic travel time tomographies in interactive 3D environments and additional scientific visualization efforts. Worked on optimization techniques by developing parallel algorithms for graphical processing unit devices applied to seismic tomography smoothing algorithms. Aided in the development of Meta-Share, a part of the center's cyber infrastructure efforts for research data management.

Professional Presentations

“Photorealistic Background Rendering” by Ivan Gris – July 2009.

Organized and presented several hands-on workshops to several college students on how to create and render photorealistic scenery.

Publications and Papers

Gris, I., and Romero, R., 2009, “Serious Games for Seismic Travel Time Tomography”, CAHSI 2010.

Chacon, C., Gris, I., and Romero, R., 2011, “Measuring and Visualizing Convergence using Cell Weighted Relative Residuals (draft)”

Permanent address: Constitucion #220 N

 Ciudad Juarez, Chih, 32000

This thesis was typed by Ivan Gris Sepulveda.