Algorithms for the Visualization and Simulation of Mobile Ad Hoc and Cognitive Networks

A Thesis
Submitted to the Faculty
of
Drexel University
by
Daniel Hennessey
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
June 2009
Acknowledgments

We would like to thank Dr. Chaomei Chen of Drexel’s iSchool for providing the citation graph data set for our graph simplification work. We would also like to thank Alex Fridman of Drexel’s ECE Department for his support in OMAN related research and Boris Shishkin of Drexel’s ECE Department for his support with the GPU based spectrum sensing work. This work is funded by the U.S. Army Communications–Electronics Research, Development and Engineering Center (CERDEC), under contract #DAAB-07-01-9-L504.
# Table of Contents

List of Tables ................................................................. v
List of Figures .................................................................. vi
Abstract ........................................................................... ix

1. Introduction ................................................................. 1

2. Mobile Ad Hoc Network Visualization ................................. 3
   2.1 Introduction .............................................................. 3
   2.2 Related Work ............................................................. 3
   2.3 Signal Radiation Pattern Visualization ............................... 4
      2.3.1 Network Model ..................................................... 4
      2.3.2 Visualization ......................................................... 5
   2.4 Digital Terrain Visualization .......................................... 8
      2.4.1 Digital Terrain Library ............................................ 8
      2.4.2 Integration with OMAN .......................................... 16
   2.5 Conclusion ................................................................ 21
   2.6 Future Work ............................................................. 22

3. A Simplification Algorithm for Visualizing the Structure of Complex Graphs 23
   3.1 Introduction .............................................................. 23
   3.2 Related Work ............................................................. 24
   3.3 Simplification Algorithm ............................................. 26
      3.3.1 Importance Metrics .............................................. 27
      3.3.2 Graph Pruning .................................................... 29
   3.4 Test Data ................................................................. 31
   3.5 Metric Performance Criteria ....................................... 32
   3.6 Results .................................................................... 32
   3.7 Conclusion ................................................................ 45
4. GPU Based Algorithms for Spectrum Sensing in Cognitive Radio Networks
   4.1 Introduction
   4.2 Related Work
   4.3 Why Use the GPU?
     4.3.1 Application to Cognitive Networking
     4.3.2 Availability and Cost Effectiveness
   4.4 Spectrum Sensing Algorithms
     4.4.1 Energy Detection
     4.4.2 Cyclostationary Spectral Analysis
   4.5 Experimental Setup
     4.5.1 Test Platform
     4.5.2 Test Data
   4.6 Results
     4.6.1 Energy Detection
     4.6.2 FFT Accumulation Method
   4.7 Conclusion
   4.8 Future Work
5. Conclusion
Bibliography
List of Tables

3.1 This table lists the Weight Range to # of Nodes ratios for each metric and each graph they were run against.............................................. 33

4.1 Timing results from Matlab execution of Welch’s Method of averaged periodograms for estimation of PSD. Timings showed for varying segment and overlap sizes. ................................................................. 62

4.2 Timing results from CUDA execution of Welch’s Method of averaged periodograms for estimation of PSD. Timings showed for varying segment and overlap sizes. ................................................................. 63

4.3 Times speedup of CUDA over Matlab for each segment size and overlap percentage. ........................................................................ 63

4.4 Break down of the timings of the Matlab based FAM code. N indicates the size of the signal being processed. All times given are in milliseconds. . 65

4.5 Break down of the timings of the CUDA based FAM code. N indicates the size of the signal being processed. All times given are in milliseconds. . 66

4.6 Shows how many times faster the GPU executed the FAM algorithm than the CPU. ................................................................................. 66
List of Figures

2.1 Signal radiation patterns for the two channel state models, (a) fully known channel state, (b) highly unstable channel state. ........................................................................ 6

2.2 Simplified versions of the signal radiation pattern visualizations for the two channel state models, (a) fully known channel state, (b) highly unstable channel state. ..................................................................... 7

2.3 Acceptable SINR regions with signal radiation patterns for the two channel state models, (a) fully known channel state, (b) highly unstable channel state. ........................................................................ 8

2.4 Acceptable SINR regions with signal radiation patterns for the two channel state models, (a) fully known channel state, (b) highly unstable channel state. ................................................................. 9

2.5 UML diagram of the design of the terrain library. ........................................ 9

2.6 The circle represent discrete points in the terrain grid. The solid box is the users original desired bounding box. The box is enlarged to the nearest discrete points that still encompass the original set of desired data. .......... 11

2.7 Example of the results produced by a horizontal profile of a region of terrain. The left side is the original terrain visualized in 3D. The right side are the results of applying the horizontal profile method. .............. 12

2.8 Figure (a) shows a sample of terrain visualized without having it lakes detected. Figure (b) then shows the same terrain with its lakes identified and visualized. ....................................................................... 13

2.9 DTED point interpolation: circles are actual data points, diamonds are interpolated points. After interpolation the user has much more data to work with................................................................. 14

2.10 DTED file ordering ................................................................................. 16

2.11 These figures illustrate 4 of the different possible color maps that can be applied to the visualization of the terrain in 2D. (a) is a standard terrain color gradient, (b) is a forest gradient shifting from white to green, (c) is a standard gray scale gradient, and (d) is an HSV color gradient starting at blue and sweeping to red. ........................................................................... 17
2.12 Figures showing OMAN network graphs visualized on top of terrain........ 18

2.13 Terrain visualization in 3D, overhead view .................................. 19

2.14 Figures showing a zoomed in view of portions of 3D terrain with portions of a physical graph visualized on top of them. .............. 20

2.15 (a) shows the nodes and edges at normal size. (b) shows the nodes and edges scaled up for better visibility. ...................... 21

3.1 Two graphs showing the node drop off rate of each metric against the Citation graph [7] (top) and an Inet graph (bottom). The min and max weight were determined for each metric on the particular graph. This range was then divided into equal increments and the simplification process was run at each increment with the threshold value being equal to each increment. The number of nodes remaining at each increment was recorded. .......... 34

3.2 Inet graph with 4500 Nodes: 2-Ring Neighbors metric with threshold values (a)300, (b)700, (c)1200, (d)1800.................................. 35

3.3 Citation Graph with 1025 Nodes: 3-Ring Neighbors metric with threshold values (a)360, (b)504, (c)576, (d)648. .............................. 36

3.4 Inet graph with 4500 Nodes: Shortest Paths metric with threshold values (a)1000, (b)40000, (c)100000, (d)645000. .......................... 37

3.5 Citation Graph with 1025 Nodes: Shortest Paths metric with threshold values (a)1404, (b)4212, (c)28828, (d)48828. ...................... 38

3.6 Inet graph with 4500 Nodes: Eccentricity metric with threshold values (a)7, (b)8, (c)10, (d)12. ................................................. 40

3.7 Citation Graph with 1025 Nodes: Eccentricity metric with threshold values (a)10, (b)13, (c)15, (d)19. ........................................... 41

3.8 Inet graph with 4500 Nodes: Shortest Distance to a Center Node metric with threshold values (a)3, (b)4, (c)5, (d)6. ...................... 42

3.9 Citation Graph with 1025 Nodes: Shortest Distance to a Center Node metric with threshold values (a)5, (b)6, (c)7, (d)8. ...................... 43

3.10 Inet graph with 4500 Nodes: Shortest Path to a Leaf Node metric with threshold values (a)3, (b)5, (c)6, (d)7. ......................... 44
3.11 Citation Graph with 1025 Nodes: Shortest Path to a Leaf Node metric with threshold values (a)2, (b)5, (c)6, (d)9. ................................................. 45

3.12 Inet graph with 4500 Nodes: Shortest Path to a Leaf Node metric with threshold values (a)0, (b)6000, (c)18000, (d)50000. ................................................. 46

4.1 Basic diagrams comparing the differences in the architectures between a CPU and a GPU. The GPU clearly devotes more transistors to data processing [29]........................................................................ 50

4.2 Graph showing a comparison of growth in peak performance between GPU’s and CPU’s over a given time line [29]......................................... 51

4.3 Graph showing a comparison of computations times for FFTs of varying sizes in Matlab and on a GPU. ....................................................... 52

4.4 Two possible implementations of an energy detector. (a) uses an analog pre-filter and square-law device. (b) uses periodogram: FFT magnitude squared and averaging. Image sourced from [6] ................................. 55
Abstract
Algorithms for the Visualization and Simulation of Mobile Ad Hoc and Cognitive Networks

Daniel Hennessey
Advisor: Dr. David Breen, Ph.D.

Visualization and simulation are important aspects of most advanced engineering endeavors. They may provide important insights into the functionality and performance of a system during the design and evaluation stage of the system’s development. This thesis presents a number of algorithms and simulation algorithms that may be used for the design and evaluation of two types of engineered systems, mobile ad hoc and cognitive networks. The first set of algorithms provides signal radiation pattern and digital terrain visualization capabilities to OMAN, a mobile ad hoc network simulator developed at Drexel University. The second set of algorithms provides a more general visualization capability for displaying complex graphs. These algorithms focus on simplifying a complex graph in order to allow a user to explore its underlying basic structure. The thesis closes with a description of a GPU-based implementation of a set of spectrum-sensing algorithms. Spectrum sensing is an important functionality needed for cognitive networks. The computational speed-ups provided by the GPU implementation offer the possibility of real-time spectrum-sensing for adaptive, cognitive networks.
1. Introduction

The field of computer networking is one in constant evolution. New advanced network architectures are developed in theory but are hard to directly implement. Simulation of these advanced networks provides a powerful tool to help develop the implementation of a network architecture while also studying the results of the implementation. Visualization of the simulated network information can play a big role in our understanding of the simulation results and the networks themselves. A creative visual display of information can increase our understanding of a simulation by providing an easier means of analysis. Research into visualizing various data associated with the simulation of mobile ad hoc networks is presented in Chapter 2. Here we discuss two visualization techniques, one for visualizing network resources at a particular layer of the network, and another for providing a richer visualization of networks by introducing the use of digital terrain. We discuss the results of implementing these visualizations in a mobile ad hoc network simulator called OMAN.

Another approach to visualizing complex information is taken in Chapter 3. This chapter focuses on developing an algorithm to help visualize the structure of large complex graphs. Large complex graphs tend to have very cluttered and unintelligible visualizations simply due to the large amount information needed to be presented. Our algorithm focuses on simplifying these graphs to some basic underlying structure and then presenting this in visualization. We use a variety of different metrics for capturing and quantifying different features of the graph to visualize. This capability allows a user to interactively discover key nodes in the graph and examine their interconnectivity. We present the results of applying our algorithm to a variety of different types of complex graphs.

The visualization of data is useful, but we also provide work done on the im-
provement of the simulations. Chapter 4 provides an investigation into how we can potentially improve the computational performance of different algorithms for cognitive networking. Specifically, we explore the improvement of several well known and well defined spectrum sensing algorithms. Spectrum sensing is an important step for the proper operation of cognitive networks. These algorithms need to perform with as little latency as possible in order to provide the best experience for all users of the cognitive network. We explore the benefits of adapting spectrum sensing algorithms to a programmable Graphics Processing Unit (GPU) in order to significantly reduce their computation times. We present a comparison of execution times on a standard CPU based implementation versus those executing on a GPU based implementation.
2. Mobile Ad Hoc Network Visualization

2.1 Introduction

The process of visualizing mobile ad hoc networks is a particularly important and challenging task. This is due to the fact that these types of networks are highly dynamic and maintain no fixed structure. The simulation of these networks is a complex process that has many different layers to understand and monitor. Visualizing information about the different layers and parameters used in simulation can help improve the understanding of these networks and the simulated results.

In this chapter we present two different visualizations. First we present a method of visualizing signal radiation patterns at the physical layer of the network model. These radiation patterns originate from transmitting nodes and are used to indicate how strong their signal is in the areas surrounding them. This pattern can be affected by a number of things such as interference from other transmitting nodes in the vicinity. Properly visualizing these radiation patterns can help us understand the effects of various parameter selections on network performance.

We then present the use of digital terrain to enhance the visualization and simulation of ad hoc mobile networks. A discussion of a library for handling digital terrain data is given, as well as how this library is integrated into the mobile ad hoc network simulator OMAN. The visualization options this new library offers are also discussed.

2.2 Related Work

The technical literature on the visualization of ad hoc networks is significantly limited. To the best of our knowledge, existing work is restricted to small-scale examples that focus more on the attainment of the data that is visualized and less
on the clarity, quality, and richness of information contained in the visual output [10]. Connectivity is a critical metric for performance of ad hoc networks, and is easy to visualize, even as the size of the network scales up. Therefore, most larger scale visual information processing efforts in the networking and visualization literature are focused on connectivity, such as in [4, 41, 3, 23]. On the other hand, tools that allow for the visualization of other aspects of the communication medium can only handle small scale networks, both in terms of running-time complexity and visual complexity; [14, 28, 12] are popular examples of such tools.

The use of digital terrain data in ad hoc network simulation is becoming more popular as it increases the realism of the simulations. The work done by [26], [5], and [11] all have modules that accounts for the use of digital terrain data in computing the propagation of the radio waves in their simulations. They use digital terrain to extract terrain profiles for a given path between two nodes. These profiles are then used in computing their propagation models. Unfortunately, none of these works make use of this terrain data for effectively visualizing the results of their simulations. The focus and novelty of the visualization methods presented here are in the effective visual representation of network resources and simulation results.

2.3 Signal Radiation Pattern Visualization

2.3.1 Network Model

The main goal of this task was to provide a visualization for the physical (PHY) layer of the networking model as defined by [46]. The PHY layer has two parameters associated with it. First, is the fixed transmission power $P_i$ on each node $i$. Second, is the minimum acceptable signal-to-interference-plus-noise (SINR) ratio, $\gamma$. The SINR
value from transmitter $i$ to receiver $j$ in time slot $z$ is given by

$$SINR_{ij}(z) = \frac{P_i A_{ij}}{\sum_{k \in \mathcal{V}_z / i} P_k A_{kj}} + \eta, (i, j) \in E^z_P, z = 1, \ldots, m$$ \hspace{1cm} (2.1)$$

where $\eta$ is the channel noise power, $E^z_P$ is the set of edges in the flow graph $G^z_P$ at time slot $z$ and power $P$, and $m$ is the number of time slots available for transmission.

The channel attenuation between transmitter $i$ and receiver $j$ is

$$A_{ij} = \left( \frac{d_{ref}}{d_{ij}} \right)^\alpha$$ \hspace{1cm} (2.2)$$

where $d_{ij}$ is the distance between nodes $i$ and $j$, $d_{ref}$ is a channel reference distance, and $\alpha$ is the attenuation exponent. All edges $(i, j)$ in each flow graph $G^z_P$ must satisfy the requirement that $SINR_{ij}(z) > \eta$.

### 2.3.2 Visualization

This model defines the parameters that are then visualized to show the signal radiation patterns from transmitting nodes. This visualization takes into account two different models of the channel. The first is a channel state that is assumed to be fully known with no fading, shadowing, or noise in the computation of the attenuation. The second model is that of a highly unstable state, modeling uncertainty in the attenuation exponent, shadowing, fading, node positions, and noise.

Figures 2.2 show us a simplified version of those visualization presented in Figures 2.1. This simplified version has removed all the receiving nodes and their associated edges. The arena size has also been zoomed into so we can more clearly see the actual patterns visualized. In analyzing the patterns of each transmitter, we can see that the patterns of neighboring transmitters can bleed into or merge with one another. This has a negative effect on the SINR of each transmitters signal.
The signal radiation patterns are shown by the change in color gradient as the signal gets weaker. The gradient starts with white as the highest value and shifts into blue for intermediate values and finally to black for a zero valued signal. This coloring scheme is easily achieved by using the HSL color scheme. The hue value can be modified to adjust the coloring of the radiation auras according to what the user may want to see. The saturation value remains a constant and the lightness values is adjusted by the visualization to perform the color fading. Figure 2.1(a) illustrates the fully known channel state model while Figure 2.1(b) illustrates the highly unstable channel state model. These two visual presentations of the physical state help demonstrate that it is much more difficult to make resource allocation decisions under uncertainty.

We can also visualize the regions where the SINR is acceptably high for a feasible channel to exist. That is, if a transmitter were to move into one of those regions, it would be able to transmit a signal to the receiving node associated with that region.
Figure 2.2: Simplified versions of the signal radiation pattern visualizations for the two channel state models, (a) fully known channel state, (b) highly unstable channel state.

These areas are represented by bright green regions in the visualization. These regions can have their transparency adjusted so the underlying signal radiation patterns can still be seen as desired. Figures 2.3(a) shows us these SINR regions in the fully known channel state model and Figures 2.3(b) shows us these same regions in the highly unstable channel state model. In Figures 2.4 we again see a simplified version of the visualization containing the regions of acceptable SINR. In these images we can clearly see how the proximity to other transmitters has an effect on the region of acceptable SINR. Although the signal strength appears to be strong enough based on the just the radiation pattern, we can now see the negative effects neighboring transmitters can have on one another.

In visualizing these regions of acceptable SINR we can also see that the presence of uncertainty makes it difficult to design node movement and scheduling such that the receiver nodes can remain in the feasible regions of the transmitter with whom they seek to communicate.
2.4 Digital Terrain Visualization

The inclusion of digital terrain data in ad hoc network simulation is not new. Terrain data is very commonly used while computing radio wave propagation models when simulating wireless networks. For this reason the effort was made to integrate the use of digital terrain data into the OMAN ad hoc wireless network simulator. The use of this data was primarily for use in more detailed calculations for running simulations, but also provides a more detailed visualization of the final results. The following sections describe the efforts taken to create a system to manage terrain data transparently and flexibly and the results of integrating that system with OMAN.

2.4.1 Digital Terrain Library

To facilitate the use of digital terrain data in the OMAN ad hoc network simulator, a library was developed to encapsulate the tasks required for managing and interacting
Figure 2.4: Acceptable SINR regions with signal radiation patterns for the two channel state models, (a) fully known channel state, (b) highly unstable channel state.

with digital terrain data. This library was designed to abstract away the details of the digital terrains file format and provide a standardized interface through which the desired data can be accessed. Figure 2.5 gives a general overview of the design structure of the terrain library. The following sections will provide further details about each class.

Figure 2.5: UML diagram of the design of the terrain library.
**Terrain Class**

The Terrain class is the primary interface through which users interact with the terrain data. The class presents the data as a large matrix where each cell contains an elevation value, measured in meters from sea level. The origin of this matrix is defined to be the lower left corner. The boundaries of this matrix are defined in two ways. First the bounding box can be accessed as a set of latitude and longitude coordinates specifying the lower left and upper right corners of the bounding box. The second method is to simply return the width and height of the area covered in meters. Accessing the grid in this manner hides the details about the resolution of the terrain data since different file formats have differing resolutions.

**Accessing Terrain Elevations** Knowing these standards for referencing coordinates within the terrain classes’ grid, users can access the actual elevation data in several different ways. The first method allows them to sample a single point of elevation at any coordinate within the defined grid. If the position they chose to sample does not contain a discrete elevation value, one will be interpolated using bilinear interpolation.

The second method of access allows the users to extract a sub-matrix of elevation data. The user specifies the lower left and upper right coordinates of the bounding box of the desired sub-matrix. If the coordinates of the users bounding box does not fall onto discrete values in the elevation matrix, the bounding box will be enlarged to the nearest set of discrete points that encompass the users original coordinates. Figure 2.6 provides an illustration of this mechanism. Using this method of access, no interpolation will be done, only discrete elevation values will be returned.

The third method of access allows the users to generate terrain profiles along either a horizontal or vertical line. The vertical profile takes two coordinates that define a
Figure 2.6: The circle represent discrete points in the terrain grid. The solid box is the users original desired bounding box. The box is enlarged to the nearest discrete points that still encompass the original set of desired data.

straight line through the terrain and the number of points the user wants returned along this line. A vector of elevation values is then returned with the desired number of points. Elevation data points are interpolated as needed to fill out the desired sized vector.

A horizontal profile is generated by defining a region of interest in the terrain by using a bounding box and giving a cutoff elevation value. This region of interest is then processed such that all points with elevation greater than or equal to the cutoff elevation remain and all others are dropped. In Figure 2.7 these points are represented by the white dots. Once this is done, polygons are created from the various connected components of these remaining elevation points. This is done using a simple convex
hull operation. In Figure 2.7 these polygons are shown as the green lines surrounding sets of white points. This set of polygons is then returned to the user.

Figure 2.7: Example of the results produced by a horizontal profile of a region of terrain. The left side is the original terrain visualized in 3D. The right side are the results of applying the horizontal profile method.

**Handling Water**  Most digital terrain file formats account for areas of water in some fashion. This terrain library maintains a map of where any water resides within any loaded terrain data. The user can then query the terrain class using an `isWater()` function for any point within the grid and get a boolean response.

In most digital terrain file formats, the elevation values are measured from sea level. Any zero valued terrain cells are then considered to be water. Identifying these bodies of water is a simple search while loading the terrain data from the files. Some digital terrain file formats, such as DTED, specify a means of identifying bodies of water that are not at sea level, such as lakes, reservoirs, or rivers. These
bodies of water must be larger than a certain size in order to be captured in the terrain data. Those bodies of water which meet the size criteria are "flattened" in the terrain data. This means that all elevation values belonging to the water body are set to the same elevation. The method of detecting these water bodies then searches for these flattened areas of terrain. The size requirements are important in order to differentiate what could be potential bodies of water from plateaus of land that may also be flat. This terrain library implements the this algorithm for identifying these bodies of water and allowing the user access to them both for computation and visualization.

Figure 2.8: Figure (a) shows a sample of terrain visualized without having it lakes detected. Figure (b) then shows the same terrain with its lakes identified and visualized.
**TerrainResolutionChanger Class**

Different digital terrain file formats deal with data of different resolutions. Some formats are very fine-grained while others are rather course. As an example, DTED files come in varying resolutions, the lowest resolution spaces elevations points every one kilometer while the highest resolution spaces points every thirty meters. In a simulation the users defined terrain region could be in an arena that spans only a few kilometers. If using very course terrain data, the number of actual data points the user would receive is very few. To help the user rectify this problem we have created a tool that allows the users to redefine the resolution of their terrain data very easily.

The user simply needs to load their desired data into our terrain class and use the *TerrainResolutionChanger* to adjust the resolution of the terrain data. The user only needs to tell the tool what they want the new spacing between adjacent elevation points to be. The *TerrainResolutionChanger* then interpolates new data points to meet this new resolution requirement using bilinear interpolation. Interpolating the values between data points offers the user much more valuable data that can be used in running a simulation. An example of this is illustrated in figure 2.9.

![DTED point interpolation](image)

Figure 2.9: DTED point interpolation: circles are actual data points, diamonds are interpolated points. After interpolation the user has much more data to work with.
For certain methods of access, the terrain class will also automatically interpolate new data as needed. The user only needs to specify the size of the desired vector or matrix they wish to fill with terrain data, and give the bounding regions of the desired area. The terrain library will then automatically fill the given vector or matrix with the necessary amount of data, interpolating data as needed.

**TerrainReader Class**

The *TerrainReader* class is what handles reading the data from specific digital terrain file formats. The *TerrainReader* makes use of a specification file in order to define what data to load and how it should be loaded. Below is a list of the definable parameters for this specification file.

- **path:** Should contain a full path to the desired terrain data files. Any standard format for specifying file paths can be used.

- **bottom_left:** Specifies the lower left corner of the desired viewport into the given terrain set. This is specified in latitude and longitude coordinates in the format \textbf{W76.6 N39.9}.

- **top_right:** Specifies the upper right corner of the desired viewport into the given terrain set. This is specified in latitude and longitude coordinates in the format \textbf{W76.6 N39.9}.

All files that contribute to the desired viewport will be loaded in their entirety. Each file is first loaded into a *TerrainPatch* object, which are simple temporary containers for the file data. As each file is loaded its *TerrainPatch* is added to a *Terrain* object. Once all files have been loaded, the *Terrain* object performs a sorting of all the *TerrainPatch* objects based on their latitude and longitude. Figure 2.10 illustrates the proper sorted order of an example set. Once sorted, the *TerrainPatch
objects are collapsed together into a single matrix within the *Terrain* object, this removes all boundaries that once existed between adjacent files. If a bounding box was defined in the specification file then this matrix is then trimmed down such that only data that belongs to that desired bounding box remains in memory.

![Diagram of DTED file ordering](image)

**Figure 2.10: DTED file ordering**

### 2.4.2 Integration with OMAN

The design of the terrain library allowed for easy integration with OMAN. OMAN is able to use the terrain library to read in specific terrain to match the desired simulation arena. The terrain can then be visualized along with the simulated network setup in either 2D or 3D. In both 2D and 3D visualizations the terrain data is turned into a height map that can be colored in a series of different color schemes. Figure 2.11 shows a sample of the different color schemes available to visualize the terrain.

Users can also choose what minimum and maximum elevation values to use when computing the color schemes. They can choose to use the current terrain’s minimum and maximum values, the known highest and lowest terrain values in the United States, or the known highest and lowest terrain values in the world. Once the terrain
Figure 2.11: These figures illustrate 4 of the different possible color maps that can be applied to the visualization of the terrain in 2D. (a) is a standard terrain color gradient, (b) is a forest gradient shifting from white to green, (c) is a standard grayscale gradient, and (d) is an HSV color gradient starting at blue and sweeping to red.

In 2D this height map is simply turned into an image and used as a background on which the network layouts can be placed on top of. Figures 2.12(a) shows an OMAN flow graph overlayed on top of a terrain region while Figure 2.12(b) shows an OMAN physical graph. Visualizing the graph structures on top of a map of the terrain can give the users more insight into how and why the simulation produced
its results. An example of this would be in a simulated movement graph, where nodes simulate intelligently moving around a given arena, the addition of visual terrain data can show whether the node movements are logical when coping with terrain obstacles.

3D Visualization Although 2D visualizations do provide useful information, looking at a height map is not always intuitive and does not always present all of the information. The visualization of terrain really benefits from 3D visualization. Just the added dimension does not provide all the potential benefits, the inclusion of interactivity in the 3D world provides the richest visualization for this data. Figure 2.13 shows a simple overhead view of a physical graph visualized on a given terrain arena in 3D. We see a graph visualized on top of a terrain arena but now with the contours of the terrain and positioning of the nodes better visualized. The 3D display highlights the separation between the nodes in the valley and those up on the right
along the right side of the arena.

![Terrain visualization in 3D, overhead view](image)

Figure 2.13: Terrain visualization in 3D, overhead view

The use of the available interactivity emphasizes the usefulness of this type of visualization. In this visualization the user has complete control and freedom of movement with the camera. Figures 2.14(a) and 2.14(b) show two different perspectives of portions of the terrain from Figure 2.13. These views are much easier to understand and comprehend than a 2d height map visualized under the plot of a graph. The features of the terrain are easily distinguishable and the effects they have on network connectivity can be easily seen when we zoom into the environment.

An example of these effects is more clearly illustrated in Figures 2.14(c) and 2.14(d). In Figure 2.14(c) we can see nodes 2 and 3 are hidden behind elements
Figure 2.14: Figures showing a zoomed in view of portions of 3D terrain with portions of a physical graph visualized on top of them.

of the terrain which may cause trouble in their connections with node 1. We see a similar situation in Figure 2.14(d). These situations are not as easily identifiable when visualized in 2D. Although one can zoom into a static 2d image, it is more difficult to interpret the exact values on a height map and try and draw conclusions that way. Seeing these situations in 3D makes it quick and easy to directly see the result.

One of the only troubles we encounter in the 3D view of this data is that when looking at the terrain as a whole it is easy for the nodes and edges to get lost in the details of the terrain. To alleviate this problem, nodes and edges can be scaled up
in size so that they are easily visible when the view is zoomed out. This feature is illustrated in Figures 2.15.

Figure 2.15: (a) shows the nodes and edges at normal size. (b) shows the nodes and edges scaled up for better visibility.

2.5 Conclusion

In this chapter we have discussed two network visualization techniques that have been implemented in the OMAN ad hoc wireless network simulator. First, we have shown a method for providing a visual representation of the signal radiation patterns of transmitting nodes. Along with this we have provided a visual representation of the regions of acceptably high SINR that allows a feasible channel to exist. Each of these visualizations are done for both certain and uncertain environments. Through these visualizations we can gain a better understanding of how parameter selection and resource allocation can affect the design of mobile ad hoc networks.

Our second visualization presented is the use of digital terrain data to improve the visualization of network simulation results. We started by discussing the terrain
library that was implemented in support of adding digital terrain data into OMAN. With that we then discussed the different 2D and 3D visualization options this then added to OMAN. Overall the addition of digital terrain to the visualization of mobile ad hoc networks adds another dimension of reality and understanding to the simulated results. We are able to better understand the effects that terrain has on the simulated networks when we are able to see the network on the actual terrain.

2.6 Future Work

In future work we hope to develop further methods of meaningfully visualizing other layers of the network model. The integration of different visualizations together would also be a worth while exploration. The display of the signal radiation patterns overlayed onto the 3D display of the network with its terrain could help to reveal further information about the simulation results. We would also like to further develop the interactivity of the 3D displays.
3. A Simplification Algorithm for Visualizing the Structure of Complex Graphs

3.1 Introduction

Graphs are used to represent and analyze a variety of data sets, from computer network topologies, to cancer cell structures, to research paper citation trends. Graphs are a useful representation for these types of data, because they naturally provide a clear and distinct way of visualizing the data sets and the interrelationships contained within them. These data sets are becoming larger and larger, and thus their visual representations are becoming too complex and cluttered. Graphs frequently have hundreds or thousands of nodes. Displaying all of these nodes and edges to a user provides no real benefit, since the density and complexity of the graph overwhelms and fills standard displays due to resolution limitations. Graph simplification provides one way of making these graphs more comprehensible by providing a capability to the user to remove “unimportant” nodes, which then exposes key nodes in the graph and their interconnectivity. This capability allows the user to interactively examine the structures exposed in simplification; thus enabling the exploration and evaluation of the graph.

In our approach a complex, cluttered graph is progressively thinned by the user until a comprehensible underlying structure in the graph is revealed. The user may choose from a variety of weighting functions that quantify the importance of the nodes to be removed. The importance metrics can be based on the topology of the graph or may be provided by an external computational process unrelated to the graph structure itself. One important feature of the simplification process is that we maintain the connectivity of the graph during pruning. By default, if the graph is
connected at the outset, we ensure that all simplified versions are also connected. The user may also choose not to employ this feature. For certain types of graphs maintaining connectivity may be important for evaluating node relationships. In other instances, allowing the graph to split into subgraphs may highlight clusters within the graph.

The simplification process involves two steps. The first step generates weight values for all nodes in a given graph based on one of our importance metrics. The second step takes the weighted graph and prunes out less important nodes based on these weight values. Node removal can be done incrementally, so as to provide different graph resolutions to the user, as well as to ensure that lower importance nodes are removed before higher importance nodes. The user may also control the connectivity maintenance feature, deciding to preserve the initial graph connectivity during simplification or not. The simplification algorithm is described in detail in Section 3.3. Results from applying several of the importance metrics to three types of graphs are presented in Section 3.6.

3.2 Related Work

Related work has been conducted on simplifying graphs, but primarily as a secondary process for manipulating or matching graphs. Qiu and Hancock [33, 32] develop three graph simplification algorithms. The first algorithm decomposes a large graph into smaller non-overlapping sub-graphs. The remaining two carry out simplification based on commute times within the graph. The algorithms produce a multilayered representation and spanning trees. All three algorithms use their simplified representations to perform matching between the input graphs.

Frishman and Tal [16] and Walshaw [42] use simplification algorithms to generate simplified graphs for input into their graph layout algorithms. The simplified graphs
are not actually visualized and presented to the user as a way to help them better comprehend the graph data. Instead, a series of progressively simplified graphs are used to guide the positioning of the nodes in a dense, complex graph.

Additional simplification algorithms have been proposed to assist in robot path planning [37], classifying the topology of surfaces [2], speech recognition [27], and improving the computational complexity and memory requirements of dense graph processing algorithms [25, 40, 13].

Simplification may also be accomplished through the use of graph clustering. Girvan and Newman [21] define one such clustering-based algorithm for better visualizing the community structure of network graphs. The kind of graphs that they are targeting with this method are those that would have a naturally occurring community structure.

Rafiei and Curial [35] present another approach to visually simplifying large scale graphs. They have developed different methods for randomly sampling a graph and using that sampling to construct the visual representation of the graph. Their general approach is similar to ours in that they are using characteristics of the graphs that frequently occur, but differs in that they reduce their large graphs down to a handful of nodes and then “grow” their simplified graph out from those nodes.

Gilbert and Levchenko [20] focus on providing metrics for simplifying graphs that represent specific network topologies. The goal of this work is to simplify and visualize complex network graphs while maintaining their semantic structures. Although network topologies are certainly reasonable candidates for these visualization techniques, there are other types of graph data that could greatly benefit from simplification techniques for visualization, e.g. the cancer cell graphs produced by Demir et al. [9] and the citation graphs of Chen [7]. These graphs do not consist of the same properties and structures as network topologies, so the metrics of [20] may not be valid for these
types of graph data. Our goal is to develop importance metrics that are based on the general topological properties of different types of graphs. This will allow for the simplification and visualization of any large graph in order for the user to better understand and comprehend its visual representation.

3.3 Simplification Algorithm

The first step of our simplification algorithm performs an importance metric calculation on the graph, generating a scalar weight value for every node in the graph. Several importance metrics have been implemented and evaluated, including number of N-ring neighbors, number of shortest paths, node eccentricity, distance to the graph’s center node(s), and distance to a leaf node. It is possible to assign importance values to the graph’s nodes based on external factors and calculations not related to the graph’s topology. In this case the calculations of the first step may be skipped, since pruning (the second step) only needs some type of importance value for the nodes, regardless of its origin.

The second step in the algorithm prunes the graph, removing nodes, and associated links, with importance values below a user-specified threshold. If desired by the user, nodes whose removal would change the graph’s connectivity are not pruned from the graph. Adjusting the threshold value allows the user to control the resolution and complexity of the simplified graph. The higher the threshold, the more nodes are removed, and the coarser the graph becomes. The user may choose to remove nodes from the graph incrementally up to the defined threshold value. Here, a number of pruning steps are applied with successively higher threshold values up to the user defined maximum threshold.
3.3.1 Importance Metrics

The weight values assigned to the nodes of a graph are computed with different importance metrics. These metrics are derived from known properties of graph structures. Currently, none of our metrics are scaled or normalized. Therefore the range of the resulting values will vary depending on the metric used, the size of the graph, as well as the topology of the graph.

Number of N-ring Neighbors

The first metric is based on the number of N-ring neighbors around a node. An N-ring neighbor is a node that can be reached in $n$ unique hops from the original node. When calculating the number of N-ring neighbors where $n$ is greater than one, the metric may be computed in two ways. The metric may count only those neighbors that lie exactly $n$ hops away from the original node. The metric may also be cumulative, i.e. it may count all the nodes that are $n$ or fewer hops away from the original node. In this paper we use the first non-cumulative option in calculating the number of N-ring neighbors greater than one.

Number of Shortest Paths

The shortest path metric is based on the graph theory principle of betweenness. A node’s betweenness is determined by how many shortest paths pass through a given node [43]. This metric finds the shortest path between all nodes $(u, v)$ belonging to graph $G$ using a variation of Dijkstra’s shortest path algorithm [22]. For a shortest path between nodes $(u, v)$, each node that belongs to the path has its weight incremented by one. The endpoints of the path do not have their weights incremented. Note that when there are two or more shortest paths of the same length between two nodes one of these is selected at random during the node increment process. This
metric is similar to the shortest path based importance metric designed by Gilbert and Levchenko [20].

**Eccentricity**

This metric assigns a node’s importance equal to the value of its eccentricity in the graph. A node’s eccentricity is defined as the maximum distance within the set of shortest paths from the node to all other nodes in the graph [22]. Higher values in this metric mean that the given node is a significant distance away from the farthest component in the entire graph, or in other words it is closer to the periphery of the graph than the center. Therefore the higher the weight value, the less important the node should be. This is opposite to the definitions of the other metrics. Thus to be consistent and to lead to proper pruning, the eccentricity importance metric is defined as $\text{eccentricity}_i - \max(\text{eccentricity})$, where $\text{eccentricity}_i$ is the actual eccentricity of a particular node $i$ and $\max(\text{eccentricity})$ is computed over all nodes in the graph.

**Shortest Distance to a Center Node**

This metric is a variation on the eccentricity metric mentioned previously. All nodes with the minimum eccentricity value are defined as center nodes of the graph [22]. These center nodes are important because they have a minimum distance to the farthest limits of the entire graph; thus placing them in the central interior of the graph. Once the center nodes of a graph have been identified, an importance metric can be computed based on the distance to one of the center nodes (there can be more than one). Nodes with shorter paths are considered more important, because they are closer to the graph’s center. As with the previous metric, this is opposite to the desired importance property. Thus the metric is computed as $\text{distance}_i - \max(\text{distance})$, where $\text{distance}_i$ is the actual shortest distance for a particular node
\(i\) and \(\text{max}(\text{distance})\) is computed over all nodes in the graph.

**Shortest Distance to a Leaf Node**

This metric calculates the shortest distance from a node to a leaf node in the graph. It is assumed that the further a node is from a leaf node, the more centrally located within the graph it is, making that node more important. Ultimately this metric assumes that the input graph will have leaf nodes. If leaf nodes are not present then this metric cannot be applied during simplification.

**Network Flow**

Importance metrics do not have to be based solely on quantities derived from the graph’s topology. They can be externally provided by other computations, e.g. network flow calculated for a graph representing a communications network. A widely-used model for ad-hoc networks is a capacitated directed graph \(G(V,E)\) [1], where an edge \((u,v) \in E\) indicates a potential communication link between transmitter \(i\) and receiver \(j\). The weights \(c(u,v)\) on the edges indicate the capacity of the transmission channel, or the maximum amount of information that can be transferred on that channel in a pre-specified amount of time. An important goal in the context of a network is to maximize the amount of information “flowing” from a source node to a destination (sink) node. A good measure of a node’s importance given this objective is the outgoing maximum flow from a node on a fully active network. The maximum flow is computed by solving a flow optimization problem on the graph [15].

### 3.3.2 Graph Pruning

The pruning process visits all nodes and removes those nodes, and their associated edges, with importance values below a user-specified threshold. Removing a node can
cause the graph to fragment. In order to maintain the basic structure of the graph we do not prune nodes whose removal would split the graph into multiple components and destroy its connectivity. Gilbert and Levchenko [20] do not maintain the connectivity of a graph during pruning. They reconnect the simplified graph as a post-process.

The connectivity check of the pruning process utilizes our variation on Dijkstra’s single source shortest path computation mentioned previously. The candidate node is removed from the graph. It is then determined if all of the candidate’s neighboring nodes are still connected to each other through the graph. This check is performed by attempting to calculate the shortest path between all pairs of neighboring nodes. If unable to complete this computation, the graph has been disconnected by the candidate node’s removal, and that node is then re-inserted into the graph.

The pruning process can be done incrementally. When pruning a weighted graph we use an increment value along with a target threshold value. The initial threshold is set to 0. The increment is added to this value and the pruning process is performed. The increment is then added to the current threshold and the pruning is performed again on the previously pruned graph. This continues until the current threshold becomes greater than or equal to the target threshold. Incrementally removing nodes will typically provide better results than a non-incremental removal. One reason for this is that it tends to remove less important nodes before more important nodes. When taking large threshold increments it is possible, since nodes are randomly accessed, that a more important node can be processed and pruned before a less important node. Removing important nodes first may leave less important nodes in a position in which they become impossible to remove due to the connectivity check. As the increment is decreased the chance of pruning a more important node before a less important node diminishes. Once the increment is equal to the smallest difference between node weight values, incremental pruning effectively produces a
removal of nodes in a sorted order, furnishing better results.

3.4 Test Data

Our graph simplification algorithm and all of its metrics have been applied to a number of graphs. The algorithm has been tested and evaluated on three different types of graphs. The first graph was automatically generated using the Inet tool [45], a power-law-based graph generator that creates graphs representing internet topologies. The Inet graph we used in our testing had 4500 vertices and 15308 edges.

The second graph was generated from data produced by the citation trends research of Chen [7]. Each node of the citation graph is a research paper and every edge represents a citation of that paper. Originally this was a directed graph, but since most of our metrics do not utilize the direction associated with an edge, we assume that the citation graph is undirected. Although we lose some of the graph’s information, it still provides a unique real-world graph structure for evaluation. This citation graph had 1025 vertices and 15430 edges.

We also include a graph generated from the OMAN ad-hoc network simulator, a simulation system developed at Drexel University. The nodes of this graph are pre-weighted using the OMAN-computed flow metrics discussed previously. We imported the graph along with its weight values and processed it with our pruning mechanism to demonstrate that the pruning operation is independent from our importance metrics calculations and is capable of handling different types of graphs.

In order to better visualize these graphs, they have each been pre-processed with the neato graph layout tool of Graphviz [36]. This applies a spring model layout to our graphs prior to applying our simplification algorithm.
3.5 Metric Performance Criteria

Once the various importance metrics were implemented and applied to the first two graphs, general properties of their resulting values were observed. In evaluating how each metric performed we first define what we believe makes an effective importance metric in terms of using it to simplify the graph for visualization.

First, a metric should provide a wide range of weight values across the nodes. A wider range allows for finer control of the resolution of the simplified graph. This property can be evaluated by the ratio of the range of weight values to the number of nodes in the graph. An example is metric \( A \) generates weights in the range of 0 to 400 on a graph \( G \) with 1000 nodes. The weight-range-to-node-count ratio would then be \( 400/1000 = 0.4 : 1 \). This provides a consistent way to compare each metric’s characteristics across a single graph. We have captured this data for all of our metrics and test graphs in Table 3.1.

An effective metric should also create an even distribution of these weight values across the entire range. An even distribution will allow for a more controlled simplification of the graph as the user increases the threshold value. This allows the user to control the resolution of the resulting simplified graph. This distribution can be measured by observing the rate at which nodes are removed from the graph while simplifying over the range of weight values.

3.6 Results

Using the metric performance criteria defined in Section 3.5, we offer an evaluation of each of our weighting metrics when applied to our test data. The distribution of the weight values across the range of weights is illustrated in Figure 3.1. The data in these graphs was generated by taking the range of the weight values of each metric on a given graph and dividing it into equal increments. Data in these graphs was...
generated by taking the range of the weight values of each metric on a given graph and dividing it into 18 equal increments. The simplification process was then run at each increment, and the number of nodes remaining after the simplification was then recorded.

N-ring Neighbors. For N-Ring neighbors we implemented three variations of the metric. We tested one, two, and three ring neighbors as metrics. Overall the N-ring neighbors metric generates reasonable weight ranges. A comparison of the weight-range-to-node-count ratios for each of the three variations can be seen in Table 3.1.

We can see that as we increase $N$, the ratios approach 1 : 1. These metrics of course are dependent on the connectivity of the original graph; but since we are primarily focusing on large complex graphs, these metrics provide good ranges for these types of graphs.

The node drop off rates for these metrics also showed a similar relationship as $N$ was increased. In Figure 3.1 the three N-Ring metrics start off with a quick drop off at the beginning of the weight range and as $N$ increases, the node drop off rate becomes approximately linear. This indicates that the distribution of weight values
Figure 3.1: Two graphs showing the node drop off rate of each metric against the Citation graph [7] (top) and an Inet graph (bottom). The min and max weight were determined for each metric on the particular graph. This range was then divided into equal increments and the simplification process was run at each increment with the threshold value being equal to each increment. The number of nodes remaining at each increment was recorded.
across the entire range becomes more evenly spread out as $N$ is increased.

Figures 3.2 show an example of the visual result generated by applying the 2-ring neighbors metric on the Inet graph. Figures 3.3 show the result of applying the 3-ring neighbors metric to the Citation graph.

![Inet graph with 4500 Nodes: 2-Ring Neighbors metric with threshold values (a)300, (b)700, (c)1200, (d)1800.](image)

Figure 3.2: Inet graph with 4500 Nodes: 2-Ring Neighbors metric with threshold values (a)300, (b)700, (c)1200, (d)1800.

Overall all the properties exhibited by the N-Ring neighbors metrics define them as effective metrics. The range of weights across the nodes and the distribution of those weight values are both in line with our definition of a good metric.
Number of Shortest Paths. The Shortest Path metric calculates very high weight values for frequently visited nodes in the center of the graph. There is a large number of paths through these central nodes since they allow the outer nodes to connect most directly with each other. This can be seen in its node drop off rate curve in Figure 3.1. Initially there is a very quick drop off of nodes. These removed nodes would exist more towards the periphery of the graph. This initial drop off tends to eliminate approximately half of the nodes in the graph within a few pruning steps. Once these nodes are removed the curve becomes much more linear in its rate of decent. This shows that the distribution of weights is very biased towards the lower end of the
The ratio of weight range to number of nodes on the other hand is quite large. For the citation graph the Shortest Path metric generated a ratio of $83.2 : 1$ and for the inet graph a ratio of $2872.6 : 1$ was generated. With such large weight-range-to-node-count ratios the difference between simplifying with two distinct threshold values may be only one or two nodes.

Figures 3.4 show an example of the visual result generated by applying the number of shortest paths metric on the Inet graph. Figures 3.5 show the result of applying this metric to the Citation graph.

Figure 3.4: Inet graph with 4500 Nodes: Shortest Paths metric with threshold values (a)1000, (b)40000, (c)100000, (d)645000.
Figure 3.5: Citation Graph with 1025 Nodes: Shortest Paths metric with threshold values (a)1404, (b)4212, (c)28828, (d)48828.

Overall the metric does not fit very well into our definition of an effective metric. The weight distribution tends to be heavily biased to the lower end of what is typically a very large range of weight values. Such a large range of weight values actually gives too much granularity between each distinct threshold value in the higher weighted nodes. The removal of one or two nodes rarely makes any distinguishable visible difference in the graphs. This metric could be made more effective by scaling or normalizing the generated weight values. Doing this should even out the distribution and contract the overly large range of weight values.
**Eccentricity.** The Eccentricity metric provides some interesting results. This metric depends on the underlying connectivity of the original graph. We can observe this by looking at the weight-range-to-node-count ratios. For both graphs this ratio we extremely low, $0.00088 : 1$ for the inet graph and $0.00878 : 1$ for the citation graph. The actual range of weight values for the inet graph was from 0 to 4. This means that the longest shortest path within this graph is length 4. This implies one of two things, that the graph is very well connected or it reveals a radial graph structure that flows into a few central nodes.

Figures 3.6 show an example of the visual result generated by applying the Eccentricity metric on the Inet graph. Figures 3.7 show the result of applying this metric to the Citation graph.

The distribution of these weight values is also highly dependent on the underlying structure of the graph. We see that for the citation graph this metric provides a relatively even drop off rate while the Inet graph has a very sharp drop off over the middle weight values in its range.

**Shortest Distance to Center Node.** Since this metric is based on the graph’s eccentricity, it displays many of the same properties as the Eccentricity metric. It maintains very low weight-range-to-node-count ratios, $0.0011 : 1$ and $0.0087 : 1$ for the Inet and citation graphs respectively, and the distribution is again dependent on the structure of the graph being processed.

Figures 3.8 show an example of the visual result generated by applying the Shortest Distance to Center Node metric on the Inet graph. Figures 3.9 show the result of applying this metric to the Citation graph.

While this metric and Eccentricity have some similarities, there are some differences in the resulting weights. In the Eccentricity metric a node relatively close to a center node may have a lower metric value because of the variety of paths available
to reach all other nodes. While that same node may have a high Center Node metric value simply because it is near a center node. In a network analogy a given node may not need to reach the edges of the network directly. Instead it connects to a more centrally located node which handles the distribution. In this case the Distance to Center metric will capture this property.

**Shortest Path to Leaf Node.** The Shortest Path to a Leaf Node metric is another that is highly dependent on the topological structure of the original graph. Firstly, it assumes that the given graph does contain leaf nodes. If it does not then this metric
cannot be computed. In relating this metric to our standards for an effective metric, this one is a bit weak. Its weight-range-to-node-count ratios are fairly small. The citation graph’s ratio was $0.0068 : 1$ while the inet graph’s was $0.00088 : 1$. These values are about the same as the Eccentricity and Distance to Center Node metrics. The node drop off rates follow a trend similar to that of the Eccentricity and Distance to Center Node metrics as seen in Figure 3.1.

Figures 3.10 show an example of the visual result generated by applying the Shortest Distance to Center Node metric on the Inet graph. Figures 3.11 show the result of applying this metric to the Citation graph.
Figure 3.8: Inet graph with 4500 Nodes: Shortest Distance to a Center Node metric with threshold values (a)3, (b)4, (c)5, (d)6.

Overall this metric is designed to give a good measure of how well connected the leaf nodes of a graph may be. It can also offer insight into how well the non-leaf nodes are connected amongst each other. If the metric’s values tend to be high, then the non-leaf nodes are not very well-connected and need to follow longer paths to get to the periphery of the graph.

**Network Flow** The computation of the Network Flow metric is one based on criteria other than the topology of the given graph. Therefore our performance criteria does not hold when evaluating this metric. We can see the results of applying the
pruning algorithm to the flow graph shown in Figures 3.12. These images present a steady thinning of the graph, while maintaining its connectivity. The $T=6000$ graph shows the drop out of all the low flow nodes in the network. The $T=18000$ graph continues to simplify the display of the network so we can start to see a more defined structure to the higher flowing nodes. The transition from the $T=18000$ graph to the $T=50000$ graph then reveals those nodes that have significant flow coming out of them, marking them as nodes which are important for flowing lots of information through the network.
Figure 3.10: Inet graph with 4500 Nodes: Shortest Path to a Leaf Node metric with threshold values (a)3, (b)5, (c)6, (d)7.

**Algorithm Run-time Performance**  The computation time needed to perform simplification is not significant. Each graph simplification requires about a second to a few tens of seconds (with a rare simplification using the Number of Shortest Paths metric needing about a CPU-minute) to compute, with most of the graph processing requiring only a few CPU-seconds on an Intel Dual Core 1.73 GHz processor. These times were to both compute the given metric and prune the graph incrementally while preserving connectivity. Times varied slightly based on the metric calculated, graph size, and threshold and increment values.
3.7 Conclusion

Complex graphs, ones containing thousands of nodes of high degree, are difficult to visualize. Displaying all of the nodes and edges of these graphs can create an incomprehensible cluttered output. We have presented a simplification algorithm that may be applied to a complex graph in order to produce a controlled thinning of the graph. The simplification of the graph provides an approach to visualizing and examining some of the underlying structures of the graph by displaying the most important nodes, where importance may be based on the topology of the graph or external factors. The simplification algorithm consists of two steps, calculation

Figure 3.11: Citation Graph with 1025 Nodes: Shortest Path to a Leaf Node metric with threshold values (a)2, (b)5, (c)6, (d)9.
of importance metrics and pruning. We have described several weighting functions and have presented the simplified graphs produced by applying them. Future work includes developing additional importance metrics, as well as testing and evaluating our approach on other types of graphs.
4. GPU Based Algorithms for Spectrum Sensing in Cognitive Radio Networks

4.1 Introduction

In a cognitive network it is assumed that a wireless spectrum is licensed to a set of primary users. These primary users have unrestricted access to this licensed spectrum. At the same time, secondary users also have access to this wireless spectrum but with the restriction that they may only use the spectrum when the primary users are not. To detect the presence of unused portions of a spectrum, these secondary users use spectrum sensing algorithms. These allow users to detect unused portions of a wireless spectrum and opportunistically occupy them until a primary user returns to use that spectrum. The idea of opportunistic spectrum sharing has seen heavy focus in the cognitive radio research community. The use of efficient spectrum sensing algorithms is necessary to facilitate this. As such, much of the research has focused on the development of these efficient spectrum sensing algorithms. Various methods have been proposed such as energy detection [6][24], cyclostationary spectral analysis [19][30], and collaborative based approaches [17][39]. Overall the computational performance of these algorithms requires improvement in order to be truly viable in order to perform real time spectrum sensing.

The goal of our work here is to implement well defined methods of spectrum sensing on a new computing architecture, a programming Graphics Processing Unit (GPU). We show that this computing architecture greatly improves the performance of these spectrum sensing algorithms both in terms of run times as well as the amount of complex computational work they can complete in a reasonable time. In Section 4.2 we discuss some of the related work around improving the performance of spectrum
This chapter is organized as follows: Section 4.3 discusses why the GPU is a viable computing architecture for improving the performance of spectrum sensing algorithms. Section 4.4 then presents the algorithms we consider for implementation on the GPU. In Sections 4.5 and 4.6 we present the testing environment, test data used, and the results gathered from the comparison of running these algorithms on the CPU vs the GPU. Sections 4.7 and 4.8 then closes with our conclusions on our findings and a short discussion of our future work.

4.2 Related Work

The literature surrounding the topic of spectrum sensing in cognitive networks is very broad. Several different works have focused on means for improving the performance of existing spectrum sensing algorithms. The work done in [24] and [34] focus on algorithms for wide-band spectrum sensing. These algorithms focus on detecting spectral holes in multiple sub-channels in a single pass of an algorithm rather than searching for spectral holes one sub-channel at a time.

Further work has focused on the development of co-operative spectrum sensing techniques, such as the work done in [17] and [39]. Co-operative based spectrum sensing aims to improve the performance and reliability of spectrum detection by having multiple nodes in the network work together in detecting spectral holes. These methods have been shown to improve performance but have not considered the possibility of using a different computing architecture to further improve performance.

The work done in [30] and [19] aimed to improve the overall performance of their spectrum sensing algorithms by porting them onto the IBM Cell processor in a Playstation 3. The architecture of the Cell processor is one meant for highly efficient parallel computation, which provided great benefit to the spectrum sensing
algorithms used in the aforementioned works. In [19] they say that the performance increase they got was limited by two factors. First the latency of transferring the data from its source to the Cell processor was significant, which hurts potential real time performance. The second limiting factor was the data rate limits of USB2, which was used in transferring the signal data from its source to the Cell processor in the Playstation 3, limiting the signal bandwidth they could work with.

Overall the literature has shown many ways in which to increase the performance of spectrum sensing algorithms. Yet no one has yet made an attempt to port these algorithms onto a General Purpose Graphics Processing Unit (GPGPU). The novelty of our work here is showing how several of these algorithms can be ported to the GPU, and to show the performance gains achieved by doing so.

4.3 Why Use the GPU?

The GPU is a highly parallelized, multi-threaded, multi-cored processor. The GPU is well suited to address problems that can be expressed as data-parallel computations - the same program executing on many data elements in parallel - with high arithmetic intensity [29]. A single GPU is capable of computing thousands of threads simultaneously with exceptionally optimized performance for high precision floating point operations. Figure 4.1 illustrates the difference in architecture between CPUs and GPUs. Since GPUs are specialized for data parallelism, they have a lower requirement for flow control and memory access latency can be hidden with high intensity calculations instead of data caches [29].

Ultimately this difference in architecture translates to higher throughput of floating point operations per second. Figure 4.2 shows a comparison of different GPU cores vs different CPU cores over a given time line. The comparison shows the growth in peak performance measured in GFLOPS. We can clearly see that GPU’s peak perfor-
Figure 4.1: Basic diagrams comparing the differences in the architectures between a CPU and a GPU. The GPU clearly devotes more transistors to data processing [29].

performance has grown significantly higher than that of any CPU currently on the market. This makes the GPU seem like the ideal platform on which to perform large scale complex computations.

Through NVIDIA’s CUDA computing engine, the massively parallel structure of GPU’s can be harnessed for general purpose computing. CUDA provides extensions to industry standard languages, such as C and Python, to access the parallel computing capabilities of NVIDIA based GPU’s. CUDA provides a set of three key abstractions - a hierarchy of thread groups, shared memory, and barrier synchronization - that are exposed to the programmer through these language extensions [29].

4.3.1 Application to Cognitive Networking

Cognitive networking algorithms rely on extremely fast computations of large sets of complex data in order to provide the necessary real time requirements for performance. In many cases, these computations can take advantage of parallel computing architectures, such as those offered by GPUs, to help increase their performance.

One of the primary operations relied upon by many of the cognitive networking algorithms is the computation of Fast Fourier Transformations (FFT) This is a common operation in many signal processing algorithms and is one that can benefit from
Figure 4.2: Graph showing a comparison of growth in peak performance between GPU’s and CPU’s over a given time line [29].

a high degrees of parallelism. Algorithms such as the FFT Accumulation Method (4.4.2) make heavy use of FFTs, so a fast implementation can make a big difference in the algorithms overall performance. Figure 4.3 shows a comparison of computation times for FFTs of complex signals of varying size using Matlab and using CUDA for GPU execution. The CUDA computations times include the necessary time to transfer the signal data from host memory (RAM) into device memory (GPU RAM). Please see section 4.5.1 for details on the testing platform used to generate these results.

In analyzing this graph we can see as the size of the FFT increases, Matlab hits a point where computations times grow exponentially while computations on the GPU
Figure 4.3: Graph showing a comparison of computations times for FFTs of varying sizes in Matlab and on a GPU.

continue to grow in a slow linear fashion. This crossover point occurs at sizes of $N = 2^{13} = 8192$ and greater. Signal data containing this many discrete points is not out of the question, especially when performing wide-band spectrum sensing.

At the same time this graph exposes a weakness of computing on the GPU. At sizes below the switching threshold, the GPU’s performance has a floor which is does not go below. This is because of the overhead associated with a) transferring the data from host memory to device memory, b) setting up the necessary framework for parallel computation. Regardless of the amount of data being processed these factors are always present. Therefore to overcome the computational overhead associated with these factors the GPU should be saturated with as much data as possible and perform computations of a complex enough nature in order to see the most benefit from its massively parallel architecture. At sizes below $N = 8192$ the GPU was simply not receiving enough data to negate the computational overhead of performing a single
FFT.

This is a fairly simplified example since we are only computing a single FFT. Typical algorithms will require much more complex computation than just a single FFT so the mentioned overhead associated with GPU computations will usually be an insignificant part of the overall computation. As such these signal size thresholds are not concrete threshold values to be targeted when considering the GPU for computing an algorithm.

Despite this weakness when using the GPU for computations, as mentioned previously, the need to process large amounts of signal data is not uncommon. Therefore when performing more wide-band operations, the GPU will likely still have a strong advantage over traditional CPU computations.

4.3.2 Availability and Cost Effectiveness

The GPU is also a very cost effective and readily available solution. Almost every desktop, and even some laptop computers, sold today have some version of a GPU residing in them. Even those desktop computers without a GPU can easily add one to its setup, and at a fairly reasonable price too.

Although cell processors are also very powerful and highly parallelizable, they are not yet as widely available as GPU’s in everyday computing devices. The only commercially available device with a cell processor is the Sony Playstation 3. It is possible to use the cell processor in the Playstation 3 for general purpose computing, as evidenced by the work done in [19], but it requires the purchase and presence of the entire Playstation 3 hardware setup. It may not be feasible or desirable to have an extra hardware piece such as the Playstation 3 tied to a real world cognitive radio. As described by [19] there is also significant latency associated with transferring the necessary data from its source to the Playstation 3 for processing. This is due in
part to the lack of a capable high speed bus through which to transfer data into the Playstation 3. GPU’s do not suffer from this problem as they have a dedicated high speed bus connection through a computer’s motherboard. From both a cost and availability standpoint, GPUs are an excellent choice for accelerating cognitive networking algorithms.

4.4 Spectrum Sensing Algorithms

The cognitive networking model assumes two kinds of users, primary and secondary users. Primary users are those who have privileged rights to a licensed spectrum for their commercial or public usage. Secondary users are those users who do not have privileged rights but still have access to licensed spectrums. These secondary users must opportunistically use unallocated spectrum resources when they are available, but vacate those resources when a primary user desires access to them. In order to facilitate the intelligent usage of unallocated resources, secondary users must have a means of detecting the unused portions of a spectrum. Spectrum sensing algorithms are designed to quickly and accurately detect the unoccupied spectrum segments available for use by secondary users.

There are several methods available to perform this kind of detection. Each method is suited to different types of environments and signal types. We have chosen to target two methods for this paper, energy detection and cyclostationary spectral analysis.

4.4.1 Energy Detection

One of the potential methods available to perform spectrum sensing is through the use of an energy detector. Energy detection is typically used to detect a weak deterministic signal in noise, which is assumed to be additive, white, and Gaussian
The energy detector works by measuring the energy in the received waveform over an observation time window [6].

Energy detectors by nature have to be suboptimal. This is due to the fact that in order to be optimal the detector would need to be based on a matched filter, thus requiring a priori knowledge of the data for coherent processing [6]. Knowing this, there are two possible ways of computing our suboptimal energy detector. The first method uses a conventional energy detector which consists of the following parts: a low pass filter, a Nyquist sampling A/D converter, and a square-law device and integrator [6]. Figure (a) provides a block diagram of this method. The second method is devised through the use of a periodogram to estimate the spectrum via the squared magnitude of the FFT of the signal [6]. Figure (b) illustrates a block diagram for this method.

Figure 4.4: Two possible implementations of an energy detector. (a) uses an analog pre-filter and square-law device. (b) uses periodogram: FFT magnitude squared and averaging. Image sourced from [6]

For our work we have chosen to work with the periodogram based energy detector described by [6]. Specifically, we have chosen the Welch method of averaged periodograms for estimating the Power Spectral Density (PSD) function as detailed
in [44]. This method breaks the signal data down into equal sized segments. Each segment is defined to be of length $M$ and we can define segments to have an overlap of $D$ samples. Each segment is then windowed, typically with a Hamming window. Following this, a modified periodogram of each segment is calculated. The computation of the modified periodogram consists of 1) taking the FFT of the segment data. 2) Finding the absolute value or magnitude of each sample and squaring it. 3) Each sample is then scaled by the value $\frac{1}{M}$. Once the modified periodogram of each segment has been computed, all the segments are averaged together to form the PSD estimate. The overlapping of the segments provides a decreased error variance in the final spectral estimate [44]. It also has an effect on the number of segments available for averaging. Greater overlap between segments means that more segments are able to fit within the same fixed signal size. Changing of the parameter $M$ also affects the error variance by increasing or decreasing the number of averages we take to generate the final spectral estimate [44]. As stated by [6], the adjustment of the segment size $M$ also has an effect on the detectors performance.

4.4.2 Cyclostationary Spectral Analysis

Another method of performing spectrum sensing is through the use of cyclostationary spectral analysis. This method is typically chosen to detect low SNR modulated signals because of its ability to distinguish between modulated signals, interference, and noise in low signal to noise ratios [18]. Cyclic spectral analysis algorithms work by estimating the correlation between spectral components of signals. Spectral components are the complex envelopes of narrow-band, bandpass components of a signal [38]. This correlation is described by the spectral correlation density (SCD) function. This function describes the localization in the frequency domain of the amount of time-correlation between frequency-shifted versions of a given signal $x(t)$ [8]. The
SCD is represented by the two dimensional cyclic frequency and signal frequency space [18] called the bi-frequency plane. Presence of a signal can then be detected through the use of the SCD function [8].

For our work we have chosen to implement the well known FFT Accumulation Method (FAM) to estimate the SCD function. The FAM algorithms is of specific interest for adaptation onto a GPU because it is able to take great advantage of data-parallelism. The following section gives details about the FAM algorithm.

**FFT Accumulation Method**

The FFT Accumulation Method works by dividing the bi-frequency plane into smaller regions called channel pair regions and computes the estimates one block at a time using FFTs [31]. The values of the SCD function are described by equation 4.1.

\[
S_{X_{N'}}(nL, f)_{\Delta t} = \sum_{r=0}^{P-1} X_{N'}(rL, f + \frac{\alpha}{2})X_{N'}^*(rL, f - \frac{\alpha}{2})
\]  

(4.1)

Equation 4.2 describes the computation of the channel pair regions, also called the complex demodulates.

\[
X_{N'}(n, f) = \sum_{n=0}^{N'-1} w(n)x(n)e^{-(i2\pi fni)/N'}
\]  

(4.2)

Here \(w(n)\) is a data tapering window (e.g., Hamming window), L is a decimation factor in the frequency domain [18], P is equal to \(N/L\) where \(N\) is the total number of samples in the signal and \(N'\) is equal to the number of samples in each complex demodulate. In choosing the value of \(N'\) we must consider that the time-frequency resolution product \(N/N'\) must satisfy the condition \(N/N' \gg 1\) in order to have a statistically reliable measurement [38]. The value of \(L\) is typically chosen to be
\( L \leq N'/4 \) in order to provide a good compromise between computational efficiency and minimizing cycle leakage and aliasing [38].

The algorithm then occurs in three steps. First the channelization of the input signal is performed by an \( N' \)-point FFT being hopped over the signal data in blocks of \( L \) samples. The results from the FFT are then frequency shifted to baseband. This produces the decimated complex demodulates [38]. Second the product sequences are computed. These are computed by multiplying each complex demodulate by the complex conjugate of each of the others [8]. Once these are computed a \( P \)-point FFT is applied to each product sequence value. The final step reorders the data from the product sequence such that it is ordered by cycle frequency vs frequency.

The parallelism of this algorithm is a direct result of the independence of the product sequences both before and after their computation [38]. This means that at each step of the algorithm, we can take advantage of high degrees of data-parallelism. In step one, each complex demodulate can be independently computed. The same can be said of computing the product sequences and applying the \( P \)-point FFTs to each of the product sequences. This very high level of data-parallelism combined with the large amount of computation necessary makes this algorithm a strong candidate for GPU based processing.

4.5 Experimental Setup

All of the algorithms were coded, compiled and tested on the platform detailed in Section 4.5.1. Timings for GPU executions were taken using CUDA supplied timing methods called from within the CUDA code. Timings in Matlab were done using the \textit{tic} and \textit{toc} function calls native to Matlab.
4.5.1 Test Platform

The primary testing platform used was a desktop computer with the following specifications:

- **CPU:** Intel Core 2 Duo E7200
  - CPU Speed: 2.53 GHz
  - Bus Speed: 1066 MHz
  - L1 Data Cache Size: 32KB x 2
  - L1 Inst. Cache Size: 32KB x 2
  - L2 Cache Size: 3MB

- **RAM:** 4GB DDR2 800 Dual Channel

- **Video Card:** Nvidia GeForce 9600 GT
  - Memory: 512MB 256-bit GDDR3
  - Memory Clock: 1900MHz
  - Clock Speed: 700 MHz
  - Stream Processors: 64

- **Operating System:** Microsoft Windows XP with SP 3

All GPU development was done using NVidia CUDA SDK version 2.1. The C++ compiler used was from Microsoft Visual Studio 2005 Professional Edition. All Matlab code was run using Matlab R2008a version 7.6.0.324. Matlab did have its multi-threading options enabled. This allows Matlab to automatically multi-thread some of its "built-in" computations such as simultaneous computation of multiple FFTs and matrix multiplication. No explicit multi-threaded programming was done within the Matlab code due to the lack of the Matlab Parallel Computing Toolbox.
4.5.2 Test Data

In testing the implementation of each of our algorithms we used a fixed set of data that we have collected. The data we use is synthetic data generated by Matlab consisting of a sinusoid plus Additive White Gaussian Noise (AWGN), sampled at 40MHz. We generated several different complex exponential signals with the sinusoid embedded at different frequencies. We generated the following signals, a sinusoid at 1MHz, sinusoid at 2MHz, and one with 3 sinusoids, one at 650KHz, one at 1MHz, and one at 3MHz. The discrete size of each signal was 16384 samples.

To make the data more realistic, it was also transmitted and received by a pair of WARP hardware nodes. WARP stands for Wireless Open-Access Research Platform, which has been developed by Rice University. This platform allow for the prototyping of advanced wireless networks. For our sample data the signals were transmitted between two WARP nodes at 2.4 GHz. During the transmission and receiving process, channel information was added into the signals, they were up converted and down converted, and some gain control was applied as well. This did have the effect of adding in some extra noise to the signals.

This test data was first used to verify the correctness of our algorithms implementations in CUDA. This data was evaluated by known working implementations of each algorithm. Results from our implementations were then carefully compared against these results to verify correctness. The data was then also used when generating the various timings for both our Matlab and CUDA based code. This step was done after correctness had been verified.

4.6 Results

This section outlines our experimental findings on the timing differences between standard CPU based implementations in Matlab versus the same implementation
done on the GPU using NVidia CUDA.

4.6.1 Energy Detection

As discussed in Section 4.4.1, we have chosen the Welch method based energy detector. We have implemented this in both Matlab and on the GPU. The following sections provide details on the implementation in Matlab and CUDA. A discussion on the results found follows.

Implementation

The implementation of our energy detector is rather simple. The Welch method is provided in Matlab as a "built-in" function called `pwelch`. Our CUDA implementation of the Welch method follows the layout of the algorithm detailed in [44].

To map this algorithm onto the GPU we first identified those portions of the algorithm that would benefit from parallel execution. This algorithm performs the same set of computations on each of the segments it creates. This provides one level of parallelism that we can exploit. The computation of the periodograms is also an operation that is highly parallelizable. The CUDA programming API provides built-in functionality for batch computing multiple FFT’s of the same size on different data sets. The squared magnitude calculations are also highly parallelizable since the squared magnitude computations are independent of one another. The averaging step of the algorithm can also benefit from parallel execution since we can compute the average of each slot in the final signal in parallel.

Having identified these regions of the algorithm for parallelization, we were then able to optimize this algorithm to run efficiently on the GPU.
The timing results from the Welch method based periodogram energy detector showed some very interesting, although not unexpected results. Tables 4.1 and 4.2 show these results. As mentioned in Section 4.4.1 varying the number of averages can improve the spectral estimate that is computed. In the following tables, $M$ is the length of each segment of the signal, which is the number of points averaged to produce the result. We also include timings of different percentages of overlay between segments to observe their affect on performance.

In looking at the Matlab timings we can see that as the segment size grows, the time for computation is reduced. This is not surprising since larger segments means fewer segments to compute, thus reducing the volume of computations necessary. We also observe that as the overlay increases the computation times rise for all segment sizes. Again this is expected since more overlay allows for more segments to be produced, thus increasing the volume of calculations necessary to form the final estimate.

In each case we can see that increasing the overlay percentage from 25% to 75% at least doubles the computation time required.

Turning to the CUDA timings we see a much different story. Unlike the Matlab timings, Table 4.2 shows very consistent timings across all variations of both segment sizes.
<table>
<thead>
<tr>
<th>Segment Size</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>25% Overlap</td>
<td>1.490ms</td>
<td>1.494ms</td>
<td>1.543ms</td>
<td>1.690ms</td>
</tr>
<tr>
<td>50% Overlap</td>
<td>1.465ms</td>
<td>1.459ms</td>
<td>1.572ms</td>
<td>1.693ms</td>
</tr>
<tr>
<td>75% Overlap</td>
<td>1.457ms</td>
<td>1.462ms</td>
<td>1.555ms</td>
<td>1.720ms</td>
</tr>
</tbody>
</table>

Table 4.2: Timing results from CUDA execution of Welch’s Method of averaged periodograms for estimation of PSD. Timings showed for varying segment and overlap sizes.

<table>
<thead>
<tr>
<th>Segment Size</th>
<th>CUDA Speedup over Matlab</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>512</td>
</tr>
<tr>
<td>25% Overlap</td>
<td>10.72x</td>
</tr>
<tr>
<td>50% Overlap</td>
<td>15.57x</td>
</tr>
<tr>
<td>75% Overlap</td>
<td>29.31x</td>
</tr>
</tbody>
</table>

Table 4.3: Times speedup of CUDA over Matlab for each segment size and overlay percentage.

size and overlap percentage. We only observe a slight increase in computation time, between 0.2 and 0.3 milliseconds, as the segment size increases. This minimal increase in computation times is negligible when compared to the fluctuation of times from Matlab.

In looking at Table 4.3 we can see the times speedup of CUDA over Matlab. We can see that the parallel architecture of the GPU allows for significant increases to the computation of the periodogram based energy detector. In the most computationally intensive setups of this algorithm, we see a nearly 30x speedup provided by the GPU.
4.6.2 FFT Accumulation Method

The following sections provides details about the specific implementation of the FAM algorithm for both the CPU and the GPU. The following is a discussion of the results found in comparing the two versions execution times.

Implementation

A full Matlab based implementation of the FFT Accumulation Method was provided by [8]. This code was used as a basis for the development of our GPU based implementation of the FFT Accumulation Method. The code from [8] was directly ported into CUDA for implementation on the GPU. Optimizations were made to the CUDA code such that we could efficiently harness the parallelized architecture of the GPU. One of the primary optimizations was the use of an existing CUDA API for performing fast FFT calculations on the GPU. In the code from [8] the computation of the product sequences and the final SCD function where both double nested for loops. Both of these sections were able to be easy optimized for parallel execution on the GPU.

We strove to maintain the same basic set of operations with the only difference between the two code bases being optimizations for running on a GPU. This was done so that we could perform a solid comparison of the effectiveness of the GPU execution times versus times on the CPU.

CPU vs GPU Timings

In Table 4.4 and 4.5 we can see the results of our timing comparisons between the two versions of the FAM algorithm. These tables give a detailed breakdown of the average time for each portion of the algorithm to compute as well as the average total time to compute the entire algorithm. Timings were done for various input signal
Table 4.4: Break down of the timings of the Matlab based FAM code. N indicates the size of the signal being processed. All times given are in milliseconds.

<table>
<thead>
<tr>
<th></th>
<th>Matlab</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>1024</td>
</tr>
<tr>
<td>Channelize</td>
<td>0.253ms</td>
</tr>
<tr>
<td>Window</td>
<td>1.042ms</td>
</tr>
<tr>
<td>Np-point FFTs</td>
<td>0.493ms</td>
</tr>
<tr>
<td>Down Convert</td>
<td>2.370ms</td>
</tr>
<tr>
<td>Product Seq</td>
<td>9.876ms</td>
</tr>
<tr>
<td>P-point FFTs</td>
<td>13.848ms</td>
</tr>
<tr>
<td>SCD</td>
<td>41.45ms</td>
</tr>
<tr>
<td>Total</td>
<td>69.432ms</td>
</tr>
</tbody>
</table>

sizes, as indicated by the $N$ values at the column headers. The CUDA results include an extra parameter in the timings, the time necessary to transfer the signal data from host memory to device memory.

In comparing the total times of execution we can clearly see that the GPU based implementation provide significant speed up. We can also observe that as the size of the input signal increases we see a substantially more significant increase in performance gained by the GPU over the CPU implementation. Table 4.6 shows how many times faster the GPU is than the CPU for the execution of the FAM algorithm at varying signal sizes.

We can also observe how different portions of the FAM algorithm benefit from the parallel architecture of the GPU. We can see that the channelization of the signal data is comparable between the two architectures. This is not surprising since the channelization process is just copying chunks of memory from one structure to another. The only case where the GPU might gain a significant advantage is when the memory architecture of the CPU is significantly slower than that of the GPU.

The application of the data tapering window shows significant differences between
Table 4.5: Break down of the timings of the CUDA based FAM code. N indicates the size of the signal being processed. All times given are in milliseconds.

<table>
<thead>
<tr>
<th></th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Transfer</td>
<td>0.564ms</td>
<td>0.544ms</td>
<td>0.537ms</td>
<td>0.58ms</td>
</tr>
<tr>
<td>Channelize</td>
<td>1.127ms</td>
<td>0.569ms</td>
<td>0.765ms</td>
<td>0.732ms</td>
</tr>
<tr>
<td>Window</td>
<td>0.51ms</td>
<td>0.404ms</td>
<td>1.238ms</td>
<td>3.343ms</td>
</tr>
<tr>
<td>Np-point FFTs</td>
<td>0.49ms</td>
<td>0.284ms</td>
<td>0.591ms</td>
<td>0.672ms</td>
</tr>
<tr>
<td>Down Convert</td>
<td>0.671ms</td>
<td>0.382ms</td>
<td>0.777ms</td>
<td>0.841ms</td>
</tr>
<tr>
<td>Product Seq</td>
<td>1.475ms</td>
<td>4.067ms</td>
<td>14.992ms</td>
<td>56.338ms</td>
</tr>
<tr>
<td>P-point FFTs</td>
<td>1.327ms</td>
<td>3.537ms</td>
<td>13.886ms</td>
<td>52.331ms</td>
</tr>
<tr>
<td>SCD</td>
<td>0.826ms</td>
<td>1.386ms</td>
<td>6.19ms</td>
<td>34.603ms</td>
</tr>
<tr>
<td>Total</td>
<td>6.993ms</td>
<td>11.179ms</td>
<td>38.98ms</td>
<td>149.445ms</td>
</tr>
</tbody>
</table>

Table 4.6: Shows how many times faster the GPU executed the FAM algorithm than the CPU.

<table>
<thead>
<tr>
<th></th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Times Speedup</td>
<td>9.928</td>
<td>25.127</td>
<td>30.429</td>
<td>31.58</td>
</tr>
</tbody>
</table>
the two architectures. This is not surprising since the application of the window is computed as a matrix multiplication operation between the channelized data matrix and a matrix with window coefficients along the diagonal. The matrix multiplication algorithm is capable of taking advantage of parallel architecture, so naturally it benefits greatly from the high levels of parallelism on the GPU.

As discussed previously in Section 4.3.1, the computation of FFTs benefits greatly from execution on the GPU. One benefit not discussed though is the ability of the GPU to batch process several FFTs of the same size in parallel. In the case of these FFT computations it is this ability that gives the GPU the edge is processing. In the case of each signal size the values of $N_p$ and $P$ are relatively small so the computation times of a single FFT would actually favor the CPU. In this algorithm it is not the size of the FFT that matters, but the number of FFTs that must be applied. The computation of the $P$ different $N_p$-point FFTs is fairly similar between the two versions. This is because the value of $P$ is typically less than 100 so a powerful enough CPU can typically keep up with the GPU in this case. We really see the GPU’s advantage illustrated when observing the time differences for the application of the $N_p^2$ different $P$-point FFTs. The $N_p^2$ term is easily in the thousands so the batch processing ability of the GPU is able to shine.

The down conversion process also shows a significant improvement in execution time on the GPU. This operation is performed as an element-wise multiplication of two matrices, an operation which again is highly parallelizable. The computation of the product sequences and the final SCD function are the two most computationally intensive portions of the algorithm, simply due to the large volume of data that needs to be processed. At all signal size values we see a significant advantage go to the GPU for these computations. Again as mentioned in Section 4.3.1, this is the kind of large scale, complex computation that the GPU excels at performing.
4.7 Conclusion

The computational complexity of various spectrum sensing algorithms can hinder their performance when computed on standard CPU architectures. Even the use of multi-core processors with multi-threaded execution can not improve the performance of these spectrum sensing algorithms as much as execution on a GPU. We have ported two standard spectrum sensing algorithms to the GPU, a periodogram based energy detector and the FFT Accumulation Method for cyclostationary spectral analysis. We have compared their computational performance when executed on a standard CPU architecture and when executed on a programmable GPU architecture. Overall we have shown that by porting these algorithms to run on a programmable GPU, the performance of these algorithms greatly improves. For both algorithms the GPU was able to provide a 30x speedup, for specific parameter setups, over the CPU based counterparts. This is due to the massively parallel computing architecture available on the GPU and the ease of access to this architecture through NVidia CUDA.

4.8 Future Work

In this chapter we have presented our initial findings in the comparison of spectrum sensing algorithms on the CPU vs the GPU. One of the main goals of our future work will be learning more advanced optimization techniques for programming on the GPU. We are still fairly new to the use of this architecture and feel that our results can still improve through further optimization of our GPU based code. Further exploration of other, more advanced, spectrum sensing algorithms, such as cooperative energy detection, techniques would prove to be an interesting study. We would also like to expand the use of our algorithms to a real cognitive radio testbed to see how well the GPU can handle live input and output in a real cognitive radio environment.
5. Conclusion

Throughout this paper we have shown a variety of results for improving the visualization and simulation of both mobile ad hoc and cognitive networks. In Chapter 2 we showed two visualization techniques useful for simulated mobile ad hoc networks. We first presented a physical layer visualization of signal radiation patterns along with regions of acceptable SINR. This visualization helped to improve the understanding of resource allocation and parameter selection on network design. Our second visualization introduced the use of digital terrain in visualizing simulated mobile ad hoc network layouts. The inclusion of terrain in the visualizations adds a level of realism and a stronger understanding of the affects that terrain can have on the performance of mobile ad hoc networks.

The work presented in Chapter 3 provided interesting results on our algorithm for visualizing the structure of complex graphs. Our algorithm showed that the simplification of the graph provides an approach to visualizing and examining some of the underlying structures of the graph by displaying the most important nodes, where importance may be based on the topology of the graph or external factors. The use of weighting metrics to isolate different types of structures should prove useful when interactively visualizing and evaluating a variety of complex graphs.

The final section of work discussed in Chapter 4, provided some very promising results on improving the performance of cognitive network spectrum sensing algorithms. We selected two well-known algorithms for spectrum sensing, periodogram based energy detection and cyclostationary spectral analysis via the FFT Accumulation Method. We were able to show that porting these spectrum sensing algorithms onto the GPU computing architecture yielded significant performance improvement, up to 30x speedup over CPU implementations.
Bibliography


