PARALLEL IMPLEMENTATION OF NONLINEAR DIMENSIONALITY REDUCTION METHODS USING CUDA IN GPU

by

Romel Campana Olivo

A project submitted in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING
in
COMPUTER ENGINEERING

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS
2011

Approved by:

_____	_____
Miguel Vélez-Reyes, PhD Member, Graduate Committee	Date
_____	_____
Nayda G. Santiago-Santiago, PhD Member, Graduate Committee	Date
_____	_____
Vidya Manian, PhD President, Graduate Committee	Date
_____	_____
Reyes M. Ortiz-Albino, Ph.D Representative of Graduate Studies	Date
_____	_____
Erick Aponte, D.E. Chairperson of the Department	Date

Abstract of Project Report Presented to the Graduate School of the University of Puerto Rico in Partial Fulfillment of the Requirements for the Degree of Master of Engineering

By

Romel Campana-Olivo

May 2011

Chair: Vidya Manian

Major Department: Electrical and Computer Engineering

Manifold learning, is one of the methods for nonlinear dimensionality reduction, which affords a way to understand and visualize the structure of nonlinear hyperspectral datasets. These methods use graphs to represent the manifold topology, and use metrics like geodesic distance, allowing embedding higher dimension objects into lower dimensional space. However the complexities of some manifold learning algorithms is $O(N^3)$, therefore they are very slow (high computational algorithms). In this project, we present a CUDA-based parallel implementation of the three most popular manifold learning algorithms: Isomap, Locally linear embedding, and Laplacian eigenmaps, using the CUDA multi-thread model. Each of these algorithms has three main parts: find K nearest neighbors, build the matrix of distances or weights, and compute the low dimension of the hyperspectral image. The first part was implemented in CUDA by (Garcia, Debreuve, & Barlaud, 2008), the second part was implemented by us in pure C++ and CUDA to measure the speedup between these implementations, and the third was carried out using the libraries of CULA and MKL

LAPACK. The manifold learning algorithms were implemented on a 64-bit workstation equipped with a quad-core Intel® Xeon with 12 GB RAM and two NVIDIA Tesla C1060 GPU cards. The CUDA implementation achieved $26x$ speedup compared to a pure C++ implementation. It also showed good scalability when varying the size of the dataset and the number of K nearest neighbors.

Resumen de Proyecto Presentado a Escuela Graduada de la Universidad de Puerto Rico como requisito parcial de los Requerimientos para el grado de Maestro en Ingeniería

Por

Romel Campana-Olivo

Mayo 2011

Consejero: Vidya Manian

Departamento: Ingeniería Eléctrica y Computadoras

Manifold learning, es uno de los métodos de reducción de dimensionalidad no lineal, el cual proporciona una manera para entender y visualizar la estructura no lineal de los conjuntos de datos hyperspectrales. Estos métodos usan grafos para representar la topología del manifold y métricas como la distancia geodésica, permitiendo representar un objeto de una dimensión mayor dentro de una dimensión menor. Sin embargo la complejidad algunos de los algoritmos de manifold learning es $O(N^3)$, por lo tanto estos son muy lentos (alto procesamiento computacional). En este proyecto presentamos la implementación en paralelo basado en CUDA de los tres más famosos algoritmos de manifold learning como son: Isomap, locally linear embedding y Laplacian eigenmaps, usando el modelo CUDA de multi-hilo. Cada uno de estos algoritmos tiene tres partes principales: encontrar los K vecinos más cercanos, construir la matriz de distancias o pesos, y calcular la dimensión baja de la imagen hiperespectral. La primera parte fue implementada en CUDA por (Garcia, Debreuve, &

Barlaud, 2008), la segunda parte se llevó a cabo por nosotros en C++ puro y CUDA para medir la aceleración entre estas implementaciones, y el tercero se llevó a cabo utilizando las bibliotecas de CULA y LAPACK MKL. Los algoritmos de manifold learning fueron implementados en un estación de trabajo de 64–bits equipado con un quad-core Intel® Xeon con 12 GB RAM y dos tarjetas GPU de Tesla C1060 de NVIDIA. La implementación en CUDA logro ser $26x$ veces más rápido que la implementación en C++ puro. También muestra una buena escalabilidad al variar el tamaño del conjunto de datos y el número de los K vecinos más cercanos.

To my God, for everything in my life,

to my family, for their unconditional support, inspiration, and love,

to my son Mikhail, for giving the reason in my life,

to my friends, for their company and their advices.

ACKNOWLEDGEMENTS

During the development of my graduate studies at the University of Puerto Rico several persons and institutions collaborated directly and indirectly with my research. Hence I wish to dedicate this section to recognize their support. Thanks to CenSSIS for supporting me during my work.

I want to express a sincere acknowledgement to my advisor, Dr. Vidya Manian because she gave me the opportunity to perform research under her guidance and supervision. I also want to thank to the members of my committee, Dr. Miguel Vélez and Dr. Nayda G. Santiago, for the opportunity of working under their supervision, support, guidance, and transmitting knowledge for the completion of my work.

Thanks to my friends, especially to Blas Trigueros-Espinosa and Samuel Rosario-Torres for their unconditional friendship and support. God bless them wherever they are.

I am grateful to my God for blessing me with the members of my church, especially to my pastors Jorge Rivera and Gorge Frankie who have made my time in Puerto Rico enjoyable and fulfilling.

Table of Contents

Figure List

Figures Page

1 INTRODUCTION

Even though hyperspectral images (HSI) have been used since the 1970s, this technology is investigated by researchers and scientists for detection and identification of materials. Hyperspectral imaging technology has applications in many fields, including agriculture, archeology, biology, defense, forensics, medicine, pharmaceuticals and remote sensing. For example, the national security system has used hyperspectral imagery to detect boats near to the coast.

Since HSI has hundreds of spectral bands and huge amount of data, traditional image processing faces difficulties. In pattern recognition and image processing, data redundancy can take two forms: spatial and spectral. Spatial redundancy is behind the spatial context methods, like morphological operations. Spectral redundancy means that the information content of a band can be totally or partly predicted from the other bands in the data. Our work consist in reduce the spectral redundancy using nonlinear dimensionality reduction methods to accelerate the further analysis of the hyperspectral data. Also, there are many linear techniques such as principal components analysis (PCA) for dimensionality reduction. PCA reduces the dimensionality of a data set by finding a new set of uncorrelated bands with high variability, smaller than the original set of bands. This technique is very popular, due to their simplicity and efficiency when the data has normal distribution. To achieve this reduction, PCA work with the correlation matrix of an image (or portion of an image); the

correlation matrix can be derived from the covariance matrix. High correlations between two bands indicate high degrees of redundancy between these two bands. However PCA assumes that the dependency between variables is linear, this can result in poor approximations when dealing with nonlinear datasets (Geiger, Urtasun, & Darrell, 2009). In some cases, PCA often obtains bad results when trying to project data lying on a nonlinear space, is desirable to reduce the dimensionality of the data while preserving the local structure of the original data, allowing for more efficient learning. Manifold learning algorithms have been developed to accomplish this task. Some of them are Isomap, Locally linear embedding, and Laplacian eigenmap. These methods are very effective when dealing with large datasets that are homogeneously sampled. "Many Manifold learning techniques provides guarantee that the accuracy of the recovered manifold increases as the number of data samples increases" (Talwalkar, Kumar, & Rowley, 2008). However, they are affected by the presence of noisy.

The complexity of some manifold learning algorithms are $O(N^3)$, which measure the running time as a function relating the number of pixels in the image "N" to the number of steps needed to solve these algorithms. For this reason, manifold learning algorithms are very slow (high computational algorithms) and computationally challenging to estimate the desired d low dimension of the hyperspectral image. The goal of this work is to parallelize the three most important manifold learning algorithms to improve the running time. All these algorithms, previously mentioned, were implemented using CUDA over GPUs (an acronym for Graphics Processing Unit), which is a parallel computing architecture developed by NVIDIA®.

1.1 Objectives

1.1.1 General objective

The overall objective is to develop a parallel implementation of the three most popular no-linear dimensionality reduction algorithms in CUDA over GPU and measure its running time. The main idea is to improve this running time, solving high computational problems in a more efficient way than on a CPU.

1.1.2 Specific objectives

The following are the specific objectives:

- Implement in parallel using CUDA, the components of the nonlinear dimensionality reduction algorithms.
- Implement in pure C++ the components of the nonlinear dimensionality reduction algorithms.
- Test the efficiency of these algorithms using different sets of hyperspectral images.
- Measure the speed up between pure C++ and CUDA parallel implementations of these algorithms.
- Change the components of the nonlinear dimensionality reduction algorithms implemented in CUDA inside the complete algorithm, to test the efficient of these algorithms.

1.2 Contribution

Nonlinear dimensionality reduction, also called manifold learning, exhibit the property that around every pixel of the manifold exits a neighborhood that is topologically the same or homeomorphic to an open subset of Euclidean space R^d (d-lower dimension). This property shows that most of the computation around each pixel based on hyperspectral imagery is using their neighborhood and it's independent of each other. Therefore, there is an inherent parallelism that can be exploited to develop new parallel algorithms of manifold learning. These algorithms are Isomap, Locally linear embedding and Laplacian eingenmaps, which are implemented using CUDA over GPU. GPU is a parallel computing architecture developed by NVIDIA.

The algorithms have been analyzed for performance - including speedup CPU versus GPU implementation of different hyperspectral images, and efficiency of the algorithms (i.e. time and utilization of parallel architectures resources). This effort was a step towards improving when the data is analyzed and allow technicians to conduct better thorough, in a timely fashion.

2 THEORETICAL BACKGROUND

In this chapter, we present a literature review of relevant concepts that we will use to develop our algorithm.

2.1 Spectral Image Basics

Hyperspectral imagery has many advantages respect other types of image like multispectral image because provides huge spectral reflectance information in hundreds of bands collected by the sensor. All materials in the earth absorb and reflect the radiation of the sun between 0.4 and 3µm spectral range. The visible light has wavelengths between 0.4 and 0.7nm (Figure 2-1).

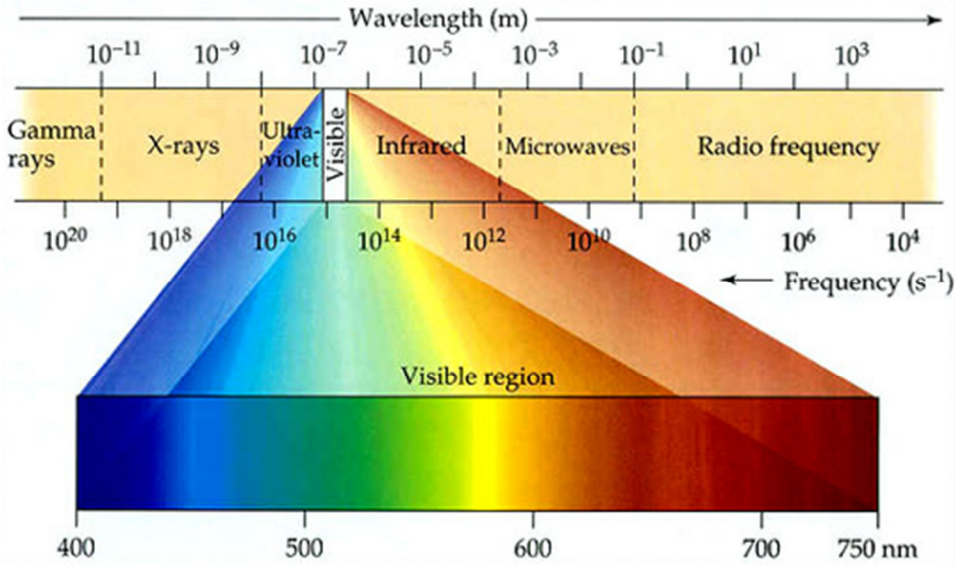

Figure 2-1: The electromagnetic spectrum (Center, 2011)

Reflectance is the percentage of the incident light in a material, which is, reflected by that material (as opposed to being absorbed or transmitted). A reflectance spectrum shows the reflectance of a material measured across a range of wavelengths, as shown in Figure 2-2. Each material on the earth has different properties that allow reflecting a wavelength, while other materials absorb the same wavelength. These material properties allow us to uniquely identify certain materials.

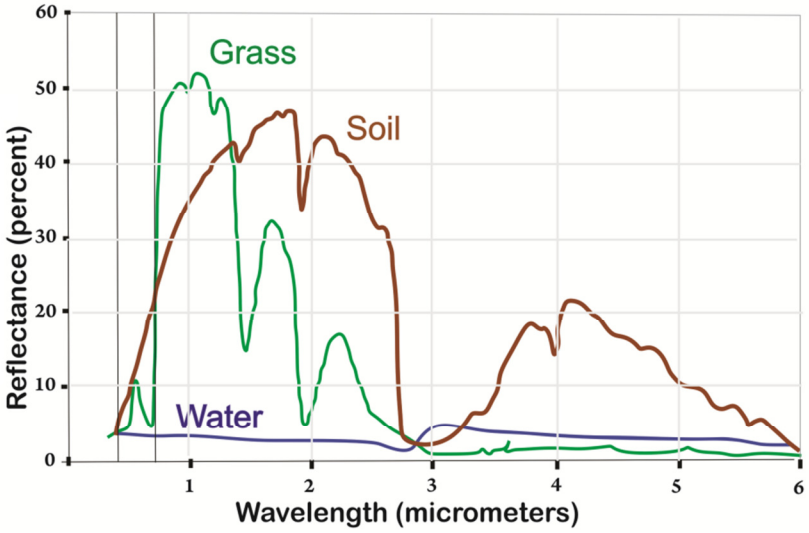

Figure 2-2: Examples of reflectance spectrum

The graph above illustrates the spectral response patterns of water, gray soil, and grass between about 0.3 and 6.0 micrometers. The graph shows that grass, for instance, reflects relatively little energy in the visible band (although the spike in the middle of the visible band explains why grass looks green). Like most vegetation, the chlorophyll in grass absorbs visible energy (particularly in the blue and red wavelengths) for use during photosynthesis. About half of the incoming near-infrared radiation is reflected, which is characteristic of

healthy, hydrated vegetation. Brownish gray soil reflects more energy at longer wavelengths than grass. Water absorbs most incoming radiation across the entire range of wavelengths. Knowing their typical spectral response characteristics, it is possible to identify forests, crops, soils, and geological formations in remotely sensed imagery, and to evaluate their condition.

2.2 Hyperspectral Imagery

The hyperspectral refers to the large number of measured wavelength bands. Hyperspectral imagery provides abundant spectral information to identify and distinguish between spectrally similar (but unique) materials. Consequently, hyperspectral imagery provides the potential for more accurate and detailed information extraction than possible with other types of remotely sensed data. However, the huge amount of data makes its information analysis difficult and image processing tools needed to summarize the information included in the data. The measurement, analysis, and interpretation of electro-optical spectra are known as spectroscopy. Combining spectroscopy with methods to acquire spectral information over large areas is known as imaging spectroscopy. Figure 3 illustrates the concept of imaging spectroscopy in the case of satellite remote sensing.

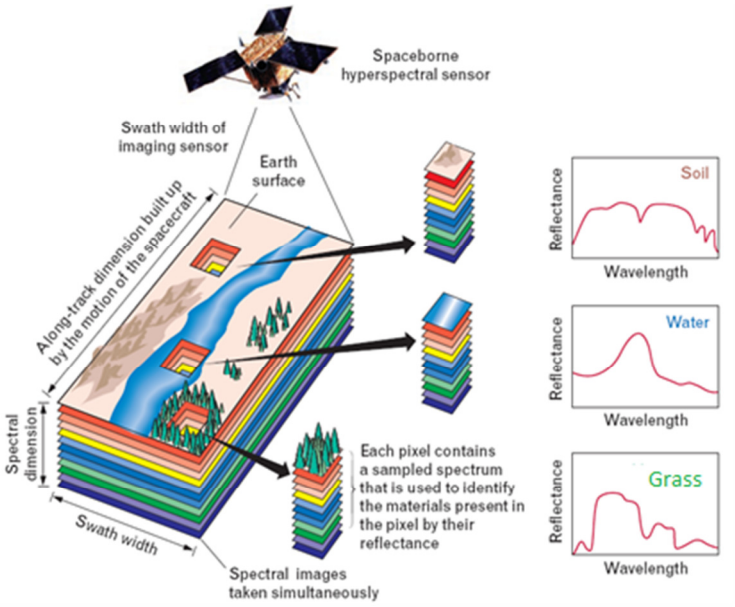

Figure 2-3: Hyperspectral imaging concept (Shaw & Burke, 2003)

Hyperspectral sensors sample the expanded reflective portion of the electromagnetic spectrum from the visible region (0.4 to 0.7 µm) through the VNIR/SWIR (about 2.5 µm). The sensors measure reflected radiation as a series of narrow and contiguous wavelength bands. When the spectrum for a single pixel in hyperspectral imagery is displayed, it can provide more information about a surface than is available in a traditional multispectral pixel spectrum. In general, most hyperspectral sensors measure bands at 10 to 20 nm of intervals in the reflective portion of the spectrum.

The detection of materials is dependent on the spectral and spatial resolution, if the pixels are too large, then multiple objects are captured in the same pixel and become difficult to identify. Another factor that affects the reliability is how much a signal has been corrupted by

noise (signal-to-noise) of the spectrometer, the abundance of the material and the strength of absorption features for that material in the wavelength region measured.

2.3 Dimensionality Reduction

Hyperspectral data sets often consist of hundreds to thousands of spectral bands, making them computationally expensive and impossible to fully visualize. Thus, it is often desirable to reduce the dimension of the data to a more manageable number of bands. One of the first steps in reducing the dimensionality of the data is to remove bands that don't carry any information about the scene or target of interest. For example, many sensors acquire data with very strong atmospheric absorption features. It is common practice to drop these bands, containing no signals from the ground, from further analysis. In addition, after atmospheric compensation (if employed), bands heavily influenced by atmospheric absorption may also be removed from further analysis. Finally, for specific targets, bands known to carry little or no information may be removed from subsequent analysis. In many cases, these processes only have a limited impact on reducing the total number of bands still needing to be analyzed. When computationally intensive algorithms are to be employed, additional dimensionality reduction may be necessary. A widely method of band combination uses the principal component method to preserve most of the variability from the original imagery in a reduced set of orthogonal bands. In this process, only the first 10-20 principal component bands are maintained and used for subsequent image analysis (Schott, 2007). This process is illustrated in Figure 2-4. In the first step, bands in the strong atmospheric absorption windows are

removed. In the second step, the remaining bands are transformed into the first d principal component bands, where the value of d is set by the user to preserve. For example, 99.8% of the variability in the data set; in fact sometimes the suggestion is that the variability must be above a threshold T, given by Equation 2.1.

$$\%\ \text{variability(d)} = 100\ \text{x}\ \frac{\sum_{i=1}^{p} \lambda_i}{\sum_{i=1}^{n} \lambda_i} \geq \text{T} \qquad \textbf{2.1}$$

Where λ_i, are the eigenvalues in descending order of the image covariance matrix and T is usually selected above 95%.

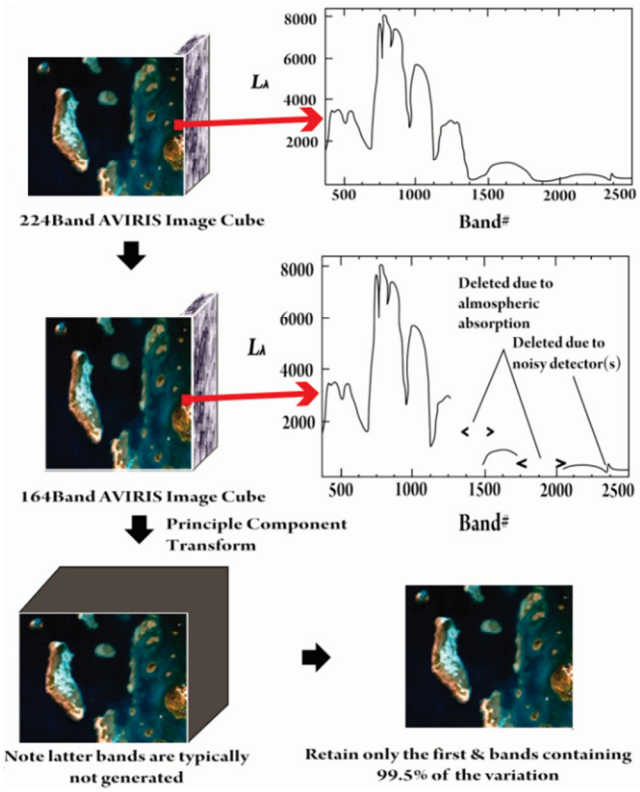

Figure 2-4: Typical steps in simple dimensionality reduction

2.4 The Manifold Learning Problem

Prior to define manifold learning problem we have to review some ideas of topology. In mathematics, topology studies the properties of objects that are preserved through deformations, twisting and stretching. Tearing is the only prohibited operation; thereby guaranteeing that the intrinsic "structure" or connectivity of objects is not altered (Lee & Verleysen, 2007)

2.4.1 A topological space

A topological space is defined as set X together with τ (the topology), as a collection of subset of X, satisfying the following axioms.

1. The empty set and X are in τ.
2. The arbitrary union of any collection of sets in τ is also in τ.
3. The intersection of any finite collection of sets in τ is also in τ.

The collection τ is called a topology on X. The elements of X are usually called points. The sets in τ are called the open sets. For example, we have a set $X = \{1, 2, 3\}$ with the topology τ, is a collection $\tau_X = \{\{\}, \{2\}, \{1, 2\}, \{2, 3\}, \{1, 2, 3\}\}$ of five subsets of X that form the topology space. An open set $X \subset \mathbb{R}$ is called open, if for each $x \in X$ there exits and $\varepsilon > 0$ such that the interval $(x - \varepsilon, x - \varepsilon)$ is contained in X, such an interval if often called an ε - neighborhood of x, or simply a neighborhood of x.

2.4.2 Manifold

First, we define formally the concept of a manifold; a manifold is an abstract mathematical space, which locally resembles the Euclidean space of a specific dimension, called the dimension of the manifold, but which globally (when viewed as a whole) may have a more complicated structure. More precisely, it is a space that can be identified locally within an Euclidean space such that the two topological spaces are homeomorphics. For example the Earth is spherical but looks flat on the human scale; the surface of the Earth is a manifold; locally it seems to be flat, but viewed as a whole from the outer space (globally) it is actually spherical (see Figure 2-5). A manifold can be constructed by 'gluing' separate Euclidean spaces together; for example, a world map can be made by gluing many maps of local regions together (Ivancevic & Ivancevic, 2007).

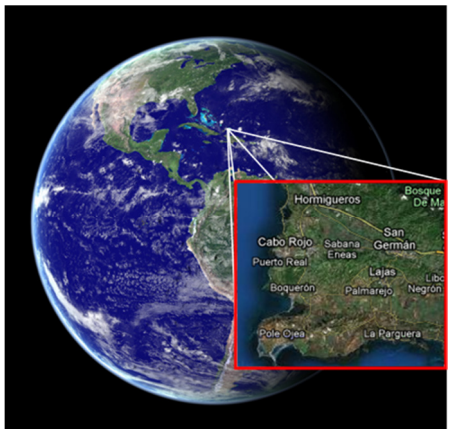

Figure 2-5: The surface of the earth is a two-dimensional manifold.

Another example of a manifold is a circle C. A small piece of a circle appears to be like a slightly-bent part of straight line segment, but overall the circle and the segment are different $1D$ manifold (see Figure 2-6). A circle can be formed by bending a straight line segment and gluing the ends together.

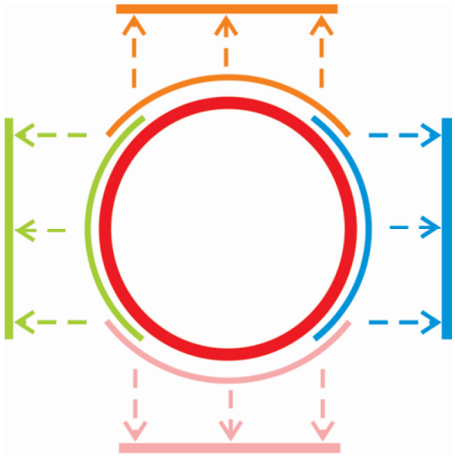

Figure 2-6: The four charts map part of the circle, and together cover the whole circle

Topology ignores bending, so a small piece of a circle is treated exactly the same as a small piece of a line. For instance, consider the top half of the unit circle, $x^2 + y^2 = 1$, where the x-coordinate is positive (indicated by the blue arc in Figure 2-6). Any point of this semicircle can be uniquely described by its y-coordinate. So, the projection onto the first coordinate is a continuous, and invertible, mapping from the right semicircle to the open interval $(1, -1)$:

$$X_{right}(x, y) = y \qquad \textbf{2.2}$$

Such functions along with the open regions they map are called charts. Similarly, there are charts for the top (yellow), bottom (red), and left (blue) parts of the circle. Together, these parts cover the whole circle and the four charts form an atlas for the circle.

Finally, consider the curve shown in Figure 2-7. The curve has intrinsic dimensionality equal to one embedded in $\mathbb{R}^3$. Note that the curve is in $\mathbb{R}^3$, has zero area. Since the curve dimensionality is one, the curve can be represented by only unique variable, because it

locally looks like a copy of $\mathbb{R}^1$. We will say that the semi circumference is a one-dimensional manifold.

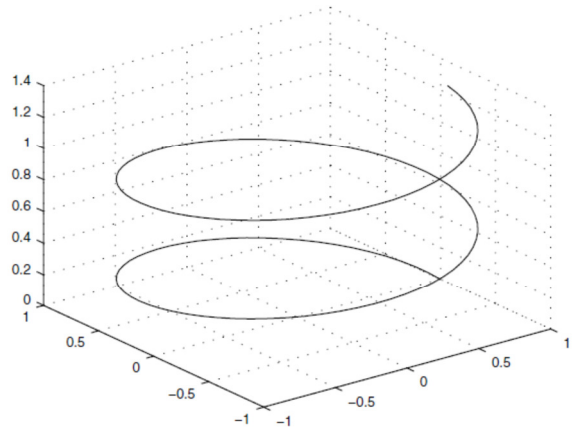

Figure 2-7: 1-D curve embedded in 3-D.

Topological manifold is a topological space which looks locally like Euclidean space. A topological space is called locally Euclidean if there is a non negative integer (greater than zero) d such that every point in X has a neighborhood which is homeomorphic to an open subset of Euclidean space in $\mathbb{R}^d$. In general any object that is nearly "flat" on small scales is a manifold.

An embedding is a representation of a topological manifold in a certain space, usually $\mathbb{R}^D$ for some D, is such a way that its topological properties are preserved. For example, the embedding of a manifold preserves open sets. More generally, a space X is embedding in another space Y when the properties of Y restricted to X are the same as the properties of X.

We pass to define formally the concept of manifold recalling the following definitions from topology. This concept can be formalized introducing the mathematical concept of a manifold (Jianru & Nanning, 2009).

Definition 1: A homeomorphism is a bijective mapping function $f: X \longrightarrow Y$ for which f and its inverse f^{-1} both are continuous.

Definition 2: A diffeomorphism exits, if two manifolds M and P has a bijective mapping function f from M to P such that both f and its inverse f^{-1} are differentiable.

Definition 3: A low d dimensional manifold M is it set that is locally homeomorphism with respecting to $\mathbb{R}^d$, in fact for each $x \in M$ an open neighborhood around x, called N_x and a homeomorphism $f: N_m \longrightarrow \mathbb{R}^d$ exists. The neighborhood N_x and the homeomorphism are respectively called the coordinate patches and coordinate chart. The image of the coordinate chart is called the parameter space.

A manifold is a very general concept. We are interested only in the special case where the manifold is a subset of $\mathbb{R}^D$, that is, $M \subset \mathbb{R}^D$ (D is the high dimension), where $d \ll D$. In other words, the manifold lies in a high-dimensional space $\mathbb{R}^D$, which is homeomorphic with a low-dimensional space $\mathbb{R}^d$.

Manifold Learning Problem Given a data set $X = \{x_1, \dots, x_N\} \in \mathbb{R}^D$ lying on a d-low dimensional manifold M described by a single coordinate chart $f: M \to \mathbb{R}^d$, find $X' = \{x'_1, \dots, x'_N\} \in R^d$ such that $x'_i = f(x_i)$, for $i = 1, \dots, N$. The solution of this problem is called manifold learning.

2.4.3 Characteristics of an analysis method

The analysis of high-dimensional data aims identifying and eliminating the redundancies among the observed variables. The principal functionality is embedding data in order to reduce their dimensionality; this yields a low dimensionality representation of data.

The knowledge of the intrinsic dimension d indicates that the data have some topological structure. The aims are both: get most data more easily. Typical applications are mostly for visualization and analysis. More exactly, if there exits an intrinsic dimensionality d; probably hide a d dimensional manifold that the manifold structure is preserved.

Figure 2-8 (a), show a two-dimensional manifold embedded in a three-dimensional space.

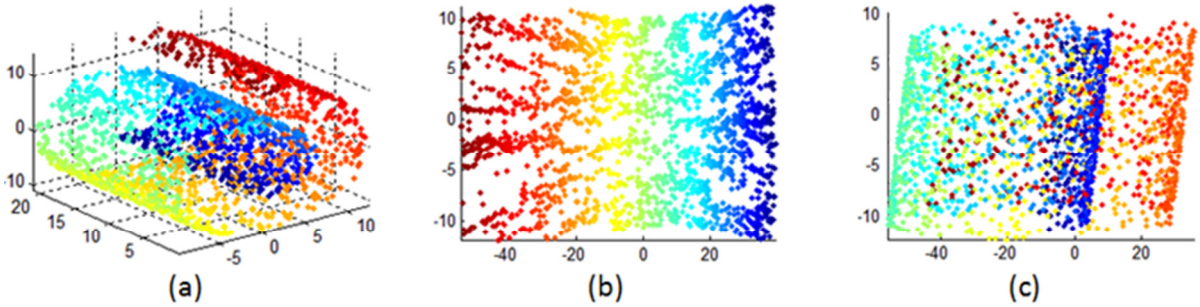

Figure 2-8: (a) A 3D space, (b) 2D manifold of (a), and (c) D.R. of (a) by PCA.

After dimensionality reduction the structure of the manifold is now completely exposed in Figure 2-8 (b) and (c). The figure show as an example view, that the nonlinear dimensionality reduction work property and preserve the structure of the objects, is important to note in Figure 2-8 (b), the connectivity and local relationship between data points are preserved.

More important, the dimensionality reduction establishes a one-to-one mapping between the three-dimensional points and two-dimensional ones. This mapping allows us to go back to the initial embedding if necessary.

PCA assumes that the dependencies between variables are linear. Therefore for this model, PCA frequently delivers bad results when trying to project the data lying on a linear subspace. This result is shown in Figure 2-8(c), where the data do not fit the model of PCA, and the initial distribution cannot be retrieval. Hence, nonlinear dimensionality reduction is preferable in these cases.

2.5 Manifold learning algorithms

Now we pass to review the most three popular manifold learning algorithms.

2.5.1 Isometric feature mapping (Isomap)

Isometric feature mapping (Isomap) was developed by (Tenenbaum, de Silva, & Langford, 2000) as a way of improving classical multidimensional scaling (MDS). Isomap is one of the most popular algorithms for nonlinear dimensionality reduction method that uses the graph distance as an approximation of the geodesic distance.

Figure 2-9 show an example of geodesic distance. In order to approximate the geodesic distance, vertices are associated with the points and a graph is built. The graph distance can be measured by summing the edges of the graph along the shortest path between both ends of

the curve. That shortest path can be computed by Dijkstra's algorithm. If the number of points is large enough, the graph distance gives a good approximation of the true geodesic distance. (Lee & Verleysen, 2007)

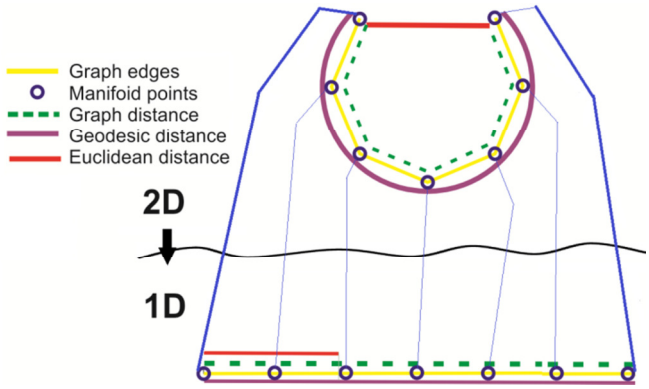

Figure 2-9: Representation of geodesic distance

Let high dimensional data set $X = \{x_1, \dots, x_N\} \in M \subset \mathbb{R}^D$ with N data samples, sufficiently large, and a low dimensional embedding space $\mathbb{R}^d$ with d << D. Isomap has the aim of finding a coordinate chart that allows projecting the data set in $\mathbb{R}^d$. Isomap assumes that an isometric chart exists, in fact a chart that preserves the distances between the points. Therefore, if two data points $x_i, x_j \in M$ have geodesic distance $GeodDist_M(x_i, x_j)$, which is the distance along the manifold, then there is a chart $f: M \to \mathbb{R}^d$ such that satisfied the Equation 2.3.

$$\|f(x_i) - f(x_j)\| = GeodDist_M(x_i, x_j) \qquad \textbf{2.3}$$

Besides, Isomap assumes the manifold M is smooth enough, such that the geodesic distance between close points can be approximated by a line. In order to compute the geodesic

distance, Isomap builds a *neighborhood graph* in the following way. Isomap computes for each data point x the set of its neighbors $N(x)$ using K nearest neighbors. After the computation of the set of neighbors for each data point x, Isomap build a graph G where each data point x is represented by a vertex in G. In addition, each vertex, corresponding to a given point x, is connected to its neighbors $N(x)$, by a weighted edge. The weighted of the edge is given by the Euclidean distances between two points, representing two vertices. Then Isomap computes the geodesic distance $GeodDist_M(x_i, x_j)$ between all data points of X by computing the shortest-path between the corresponding vertices on the graph G using Dijkstra's algorithm. At the end of this step, Isomap produces a matrix $GeodDist_M$ whose element $GeodDist_M(i,j)$ is given by the geodesic distance between the data points x_i and x_j, that is show in Equation 2.4.

$$GeodDist_M(i,j) = GeodDist_M(x_i, x_j) \quad \textbf{2.4}$$

The final step of Isomap consists in applying a MDS to construct the lower dimensional space d, which preserve as much as possible the structure of the manifold.

Isomap can be summarized in the following steps:

1. For each data point $x_i \in X$, find its K nearest neighbors.
2. Build the neighborhood graph.
3. Compute the shortest path graph given the neighborhood graph.
4. Find the d- low dimensional embedding by MDS algorithm.

The parameter K controls the size of the neighborhood and this value is crucial. Isomap is strongly influenced by the size of neighborhood.

2.5.2 Locally linear embedding

Locally linear embedding (LLE) was proposed by (Roweis & Saul, 2000). The algorithm tries to preserve the topology by keeping all the neighbors close to each other. The idea of LLE is then to replace each point x_i where $1 \leq i \leq N$, with the linear combination of its neighbors. Therefore, the local geometry of the manifold can be characterized by linear coefficients that reconstruct each data point from its neighbors. Suppose high dimensional data set $X = \{x_1, \dots, x_N\} \in \mathbb{R}^D$ with N data samples, sufficiently large, and a low dimensional embedding space $\mathbb{R}^d$ with $d << D$. The weight matrix W is computed by minimizing the reconstruction error $\varepsilon(w)$, where w_{ij} are the coefficients of each point x_i and its neighborhood.

$$\varepsilon(w) = \sum_{i}^{N} \left| x_i - \sum_{j} w_{ij} x_j \right|^2 \qquad \textbf{2.5}$$

The sum of the coefficients of w_{ij} must be equal to one: $\sum w_{ij} = 1$. The cost matrix W is expanded to a sparse NxN matrix W with $w_{ij} = 0$ if x_j does not belong to the neighborhood of x_i.

In the final step of LLE, each high-dimensional data point is mapped to a low-dimensional vector representing global intrinsic coordinate on the manifold. This is done by choosing d dimensional coordinates to minimize the embedding cost function:

$$\varphi(Y) = \sum_{i}^{N} \left| y_i - \sum_{j} w_{ij} y_j \right|^2 \qquad \textbf{2.6}$$

This cost function sums the reconstruction errors caused by locally linear reconstruction. In this case, however, the errors are computed in the embedding space and the coefficients $w_{ij}$are fixed. The minimization of the function $\varphi(Y)$ gives the low dimensional coordinates Y= $\{y_1, \dots, y_N\} \in \mathbb{R}^d$ that best reconstruct y_i given W.

The LLE algorithms is executes as follows

1. For each data point $x_i \in X$, find its K nearest neighbors.
2. Compute the weights matrix w_{ij} that best linearly reconstruct $\mathrm{x_i}$ from its neighbors and minimizes the Equation 2.5.

   The sum of the coefficients of w_{ij} must be equal to one: $\sum w_{ij} = 1$
3. Find the d-dimensional embedding vectors $\mathrm{y_i}$ by using the weights w_{ij}, which minimizing the Equation 2.6.

2.5.3 Laplacian eigenmap

Laplacian eigenmap was developed by (Belkin & Niyogi, 2003). In contrast with Isomap, this method work locally, reproducing small linear patches around each point. In that sense, Laplacian eigenmap is closely related to LLE. Given a data set $X = \{x_1, \dots, x_N\} \in \mathbb{R}^D$, the algorithm construct a weighted graph incorporating K information's neighbors, one for each point, and then uses the graph Laplacian to compute the low-dimensional representation while preserving local neighborhood information.

The algorithmic procedure is formally stated below.

1. For each data point $x_i \in X$, find its K nearest neighbors.
2. Constructing the adjacency graph, by putting an edge between nodes i and j, if x_i and x_j are neighbors.
3. Assign edge weights. The edge weights are determined using one of the following variations:
   a) Heat kernel uses a parameter $\varepsilon \in \mathbb{R}$. Specially if two nodes i and j are connected, put $W_{ij} = e^{-\frac{\|x_i - x_j\|^2}{\varepsilon}}$; otherwise, $W_{ij} = 0$.
   b) Simple-minded is getting if $\varepsilon = \infty$ in $W_{ij} = e^{-\frac{\|x_i - x_j\|^2}{\varepsilon}}$, no parameter is used. Where $W_{ij} = 1$, if vertices i and j are connected by an edge and $W_{ij} = 0$ otherwise. This simplification avoids the need to choose ε.

4. Compute Laplacian eigenmap, solving the generalized eigenvalue problem $LY = \lambda DY$, where D is diagonal weight matrix, and its entries are column (or row, since W is symmetric) sums of W, $D_{ii} = \sum_j W_{ij}$. $L = D - W$, is the Laplacian symmetric, positive semidefinite matrix.

2.6 CUDA and GPU

Compute Unified Device Architecture (CUDA) is a new parallel programming model that uses Graphics Processing Unit (GPUs) to solve different computational problems in a more efficient way than on a CPU. CUDA is accessible to software developers through variants of industry standard programming languages. Programmers use C for CUDA (C with NVIDIA library extensions and certain restrictions).

2.6.1 Hardware architecture

CUDA is all based on a common hardware architecture, which is outlined in Figure 2-10. Each device consists of a certain number of multiprocessors and its device memory. A multiprocessor consists of eight scalar processor cores, which operates in a SIMT (single instruction multiple thread) fashion, all cores in the same group execute the same single CUDA program called kernel at the same time in parallel. The device memory can be accessed by all multiprocessors of a device. A warp is 32 threads actives supported on each

multiprocessor. When one warp stalls on a memory operation, the multiprocessor selects another ready warp and switches to that one.

2.6.2 Programming model

From the point of view of a software developer, a CUDA compatible graphics card appears as a highly parallel processing unit for general purpose computations, which can be used in addition to the system's CPU. The processing unit (commonly called CUDA device) is addressed by the system's CPU (commonly called host) by methods of CUDA host API, which allows it to manage device resources and kernel invocations. This API acts as a new interface to CUDA capable graphics boards in addition to existing low-level.

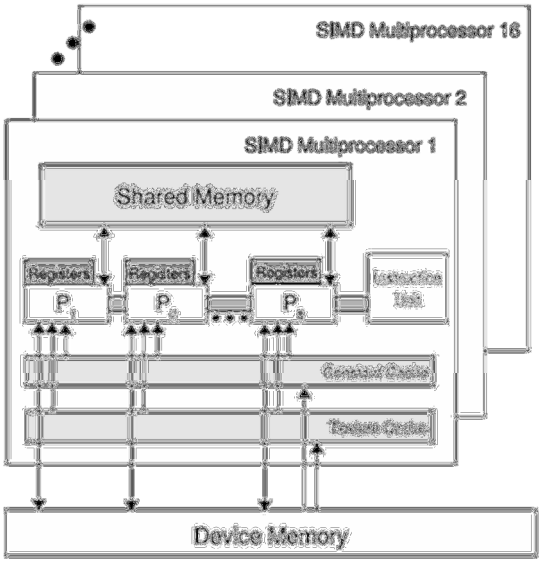

Figure 2-10: C Program sequential execution (Zone, 2011)

2.6.3 *Kernel execution*

A kernel is executed on the CUDA device by a set of threads in parallel; a user defined group of 1 to 512 (1024 in some devices) threads called blocks. The instruction $dim3\ dimBlock(x, y)$ is used to define the number of threads per block, where x is the number of columns and y is the number of rows in the block respectively. For identification and addressing purposes, each thread in a block has a unique thread index called $threadIdx$ in x and y. The $threadIdx.x$ value ranges between 0 and $blockDim.x - 1$ and the $threadIdx.y$ value between 0 and $blockDim.y - 1$. The Figure 2-11 shows, as example, the instruction $dim3\ dimBlock(7,7)$ where $x = 7$ and $y = 7$.

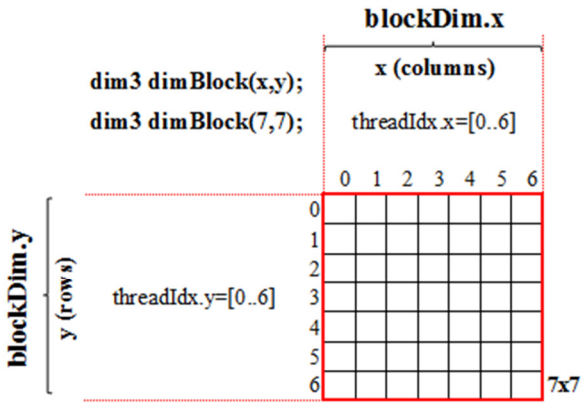

Figure 2-11: Block of threads

The second important concept is a $grid$. A grid is a set of blocks that is organized in a grid of two dimensional arrays of equally sized thread blocks. All threads in a block share the same $blockId$ value. The grid and thread blocks may be chosen as one, two or three dimensional to ease data addressing. The instruction $dim3\ dimGrid(x, y)$ serves to define the number of

blocks in a grid, where x is the number of colums and y is the number of rows respectively, each block in a grid has a unique block index called $blockIdx$ in x and y. The $blockIdx.x$ value ranges between 0 and $gridDim.x - 1$ and the $blockIdx.y$ value between 0 and $gridDim.y - 1$. The grid has a maximum in x or y (dimension) of 65535 blocks of threads. Figure 2-12 shows a grid of 4 rows and 1 column using the instruction $dim3\ dimGrid(1,4)$ where $x = 1$, and $y = 4$.

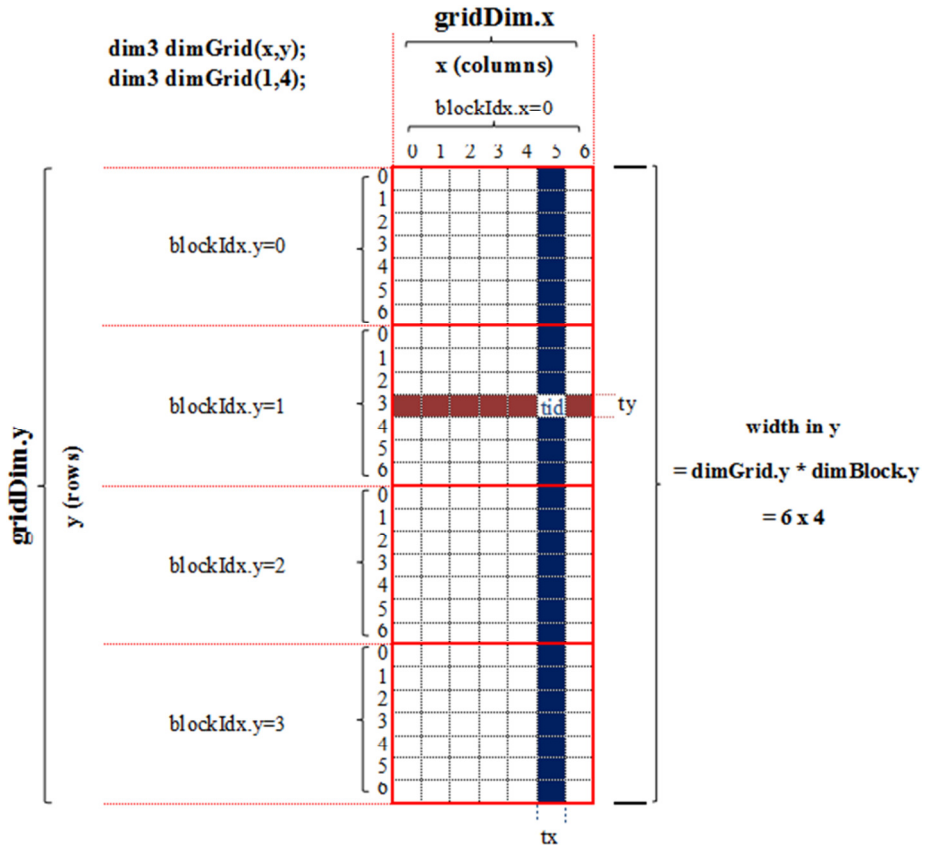

Figure 2-12: Grid of blocks of threads

The number of threads per block and the number of blocks per grid, give the total numbers of thread that will be executed on the device when a CPU invokes a kernel function. All threads

of a thread block are executed concurrently on one multiprocessor. The following instruction is an example of a kernel invocation by CPU.

```
// Invoke kernel
dim3 dimBlock(7,7); //The maximum value of K is 22 for 512 threads per block
dim3 dimGrid(1,4);
My_Kernel<<<dimGrid, dimBlock>>>(d_data, d_result, other_parameters);
```

In CUDA is very important to define the number of threads per block, because this choice affects the performance of the kernels within a CUDA program. The Figure 2-13 summarizes the above description.

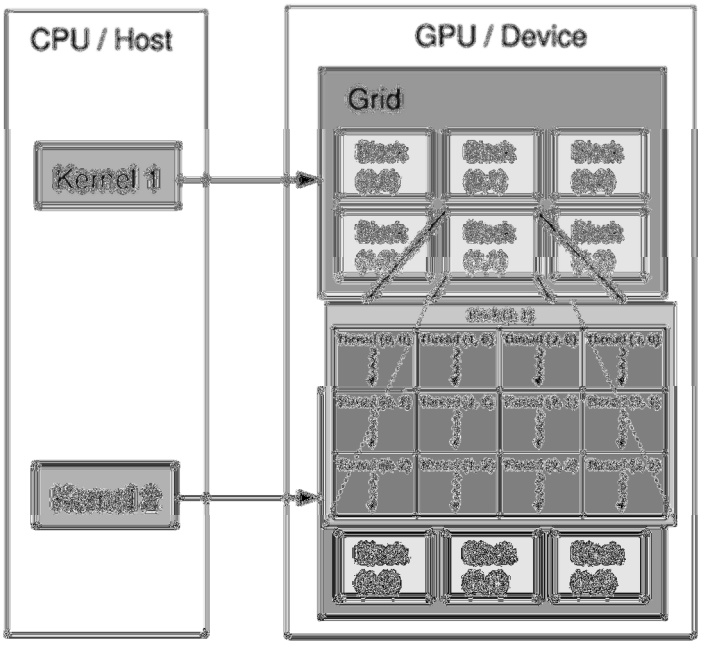

Figure 2-13: Kernel execution configuration

2.6.4 Kernel programming

When a kernel is executed, each thread in a kernel is manage within a group of 32 threads called warp, which assigns a set of registers and local memory needed for its execution.

Shared memory is accessed by all threads of a block allowing cooperation between them and the exchange of data. Shared memory is much faster than global memory. The sequence of execution of a CUDA program has serial code that will be executed on the host while parallel code will be executed in parallel on the device; see Figure 2-14.

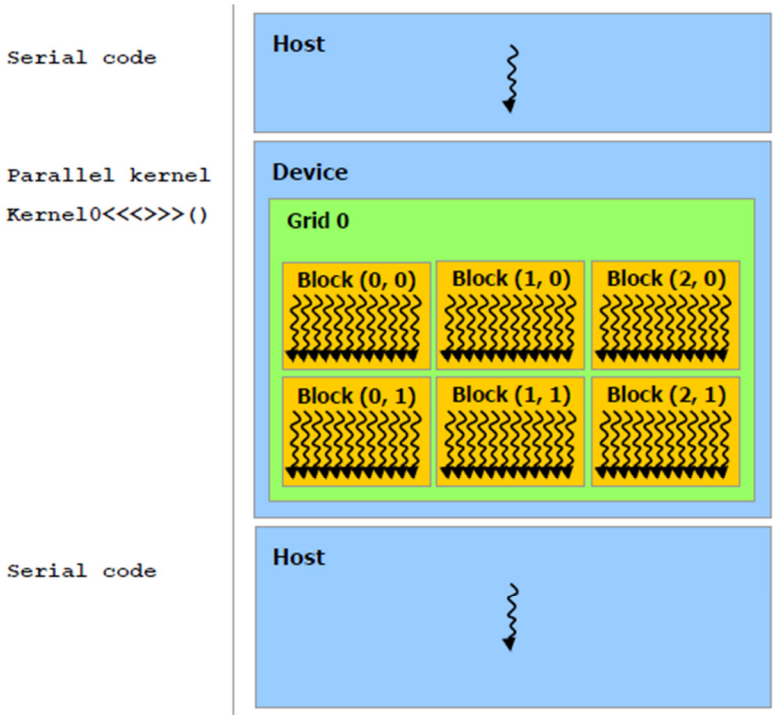

Figure 2-14: Software execution model

3 IMPLEMENTATION OF MANIFOLD LEARNING ALGORITHMS IN CUDA

The principal idea is to implement in parallel the three most popular nonlinear dimensionality reduction algorithms using CUDA over GPU to improve the running time of the algorithms and measure the speedup between CPU and GPU implementation. In addition, measure the speedup between CPU and GPU implementation. Figure 3-1, summarize this research method.

All nonlinear dimensionality reduction algorithms require to compute the Euclidean distance matrix, to select the K-nearest neighbor, this means that the number of neighbor for each data point needed in the first part of the manifold learning algorithms is given by parameter K. This parameter K is entered by the user or observer and this value influence in the results dramatically.

After the graph is built, each algorithm uses this information for specific purpose. In the case of Isomap, this compute the shortest path distances between all pair of points using Dijkstra's algorithm. LLE compute the reconstruction weights matrix, and Laplacian eigenmap build the weighted adjacency matrix. All these algorithms embed the graph built G into the low-dimensional space.

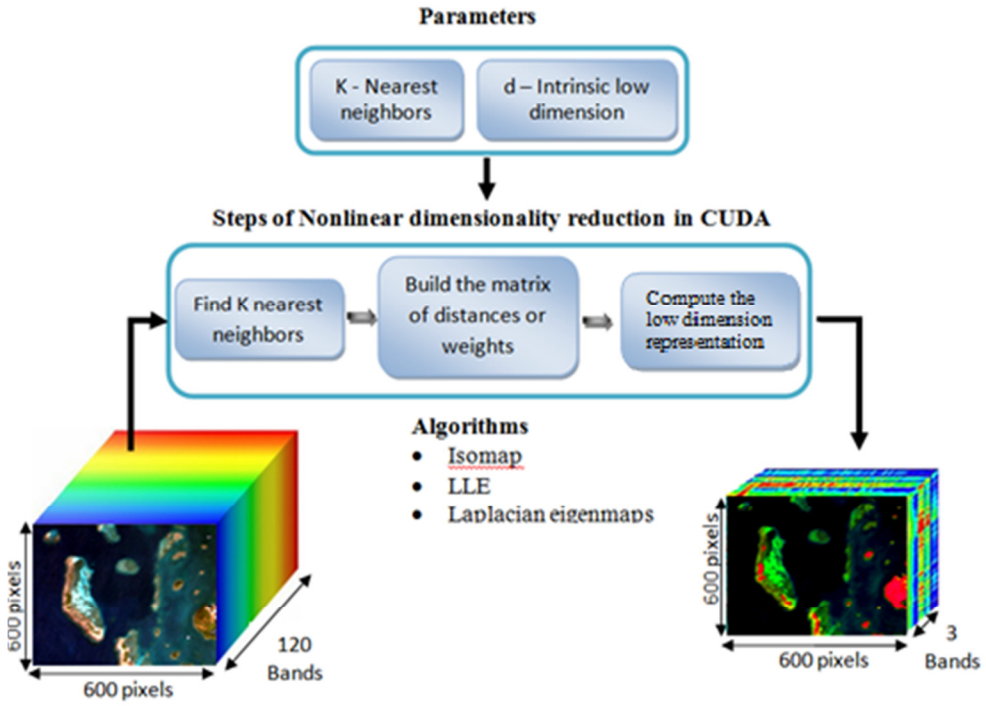

Figure 3-1: Process for nonlinear dimensionality reduction

3.1 Isomap in CUDA

The implementation of Isomap is described in Algorithm 1, for more details see Section 2.5.1. Isomap algorithm has three parameters: X is the data set of size $N \times D$, K it the nearest neighbors of x_i, and the intrinsic desired low dimension d.

The parameter X is the hyperspectral image in high dimension D, which could be any data set with any number of features; the parameter K controls the size of the neighborhood and this value is crucial. Isomap is strongly influenced by the size of neighborhood, and the last parameter d is the desired low dimension.

Algorithm 1 Isomap (X, K, d)

1: For each data point $x_i \in X$, find its K nearest neighbors.
2: Build the neighborhood graph.
3: Compute the shortest path graph given the neighborhood graph.
4: Find the d-dimensional embedding by MDS algorithm.

The line 1 of Isomap algorithm was implemented in CUDA by (Garcia, Debreuve, & Barlaud, 2008). Given a reference point set X and a query point set X (the same data set), the program returns first the *matrix of distances* between each query point and its k nearest neighbors in the reference point set, and second the *matrix of indexes* of its K nearest neighbors. The matrix of indexes is also called the neighbors of X. The graph in line 2 is implicitly represented by adjacency matrices of indexes and distance that serve to compute the shortest path graph in line 3. The implementation of line 3 was based in the previous work (Pawan & Narayanan, 2007), for a single source shortest path was modified to compute all pairs shortest path given a X data set (see Algorithm 2). Finally, the line 4 of Algorithm 1 was implemented using CULA, that is a GPU-accelerated linear algebra library that utilizes the NVIDIA CUDA parallel computing architecture to improve the computation speed of sophisticated mathematics (Photonics, 2010). The premium package provides the computation of eigenvalues and eigenvectors. The instruction used was culaSsyevx.

All pairs shortest path problem is, given weighted graph $G(V; E; W)$ with positive weights and a source S (in our approach, each entry in diagonal matrix is a source vertex), find the path with lowest cost (the shortest path) between the source vertex and every other vertex in V. The common implementation of Dijkstra's algorithm is based on a min-priority queue

implemented by a Fibonacci heap that has a time complexity $O(V^2\ logV^2\ + E)$. Now the memory space required store the cost matrix is show in Figure 3-2. If n=128, D = 64 with a single data type, the storage for X in 3D or 2D is:

$$N \times D \times 4\ bytes\ =\ 128 \times 128 \times 64 \times 4\ bytes =\ 2^{22} bytes = 4MB.$$

Now the storage of the geodesic distance matrix is $N^2 \times\ 4 = 2^{30} bytes = 1GB,$ with $N\ =\ n \times n\ =\ 128 \times 128$, therefore to build the geodesic distance of $4MB$ is required 1GB of storage. For this reason we work with images up to $n\ =\ 128$. All $3D$ X data set is reshaping in $2D$ representation and with the matrices of indices and distances of K nearest neighbors, Dijkstra's Algorithm in line 3 finds shortest path for all points.

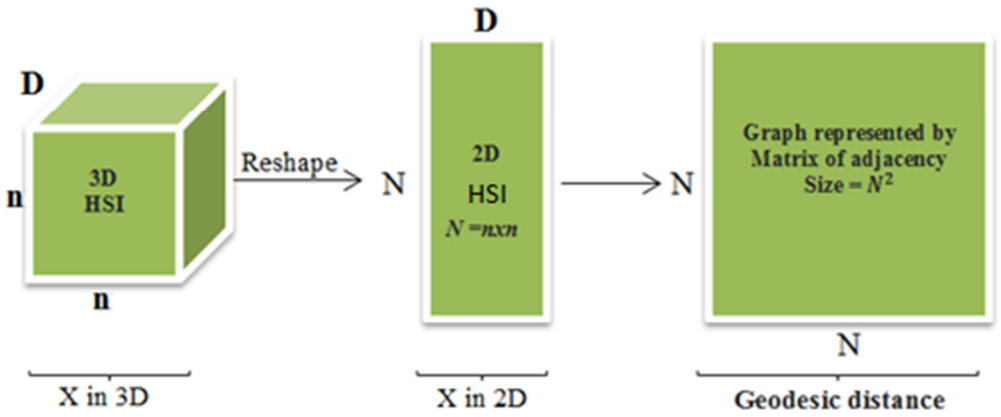

Figure 3-2 Reshaping 3D hyperspectral image in 2D and the storage required for geodesic distance.

Algorithm 2 shows a CUDA implementation of all pairs Shortest Path, the termination is based on the change in cost matrix C of size $N \times N$. In this implementation, was used a an edge matrix E that represent the matrix of neighbors or indices of size $N \times K$, a weight matrix W of size $N \times K$ that represent the matrix of neighbors's distance of size $N \times K$, a

boolean mask matrix M of size $N \times N$ and a updating matrix U of size $N \times N$. In each iteration of the Algorithm 2, Algorithm 3 CUDA Shortest Path Kernel Step 1, checks for each vertex if it is in the mask matrix M. If yes, it fetches its current cost from the cost matrix C and its neighbor's weights from the weight matrix W. The cost of each neighbor is updated if greater than the cost of current vertex plus the edge weight to that neighbor. The new cost is not reflected in the cost matrix but is updated in an alternate matrix U. At the end of the execution of the kernel, a second kernel, Algorithm 4 CUDA Shortest Path Kernel Step2, compares cost in matrix C with updating cost.

Algorithm 2 CUDA Shortest Path (Graph G (V; E; W))

1: Create the square mask matrix M, cost matrix C and updating cost matrix U of size $N \times N$
2: Initialize mask M to false, cost matrix C and Updating cost matrix U to ∞
3: $M[diagonal] \leftarrow true$
4: $C[diagonal] \leftarrow 0$
5: $U[diagonal] \leftarrow 0$
6: **while** M not Empty **do**
7: **for** each vertex V^2 **do in parallel**
8: Invoke **CUDA Shortest Path Kernel Step1** (E, W, M, C, U) on the grid
9: Invoke **CUDA Shortest Path Kernel Step2** (M, C, U) on the grid
10: **end for**
11: **end while**

Algorithm 3 CUDA Shortest Path Kernel Step1 (E, W, M, C, U)

1: tid ← get threadId
2: **if** M[tid] **then**
3: **for all** neighbors id_nb of tid **do**
4: **if** U[id_nb] > C[tid] + W[id_nb] **then**
5: U[id_nb] ← C[tid] + W[id_nb]
6: **end if**
7: **end for**
8: M [tid] ← false
9: **end for**

Algorithm 4 CUDA Shortest Path Kernel Step2 (M, C, U)

```
1: tid ← get threadId
2: if C[tid] > U[tid] then
3:     C[tid] ← U[tid]
4:     M[tid] ← true
5: end if
6: U[tid] ← C[tid]
```

3.1.1 Process for Isomap

The complete process for Isomap algorithm is shown in Figure 3-3. The user through the interface, enter the parameters X, K, and d and call Isomap for execution, the interface call to the *control program* who is responsible for executing the complete program of Isomap. The *control program* execute some parts of the program inside the CPU and another parts inside the GPU. First the program reshape the image from $3D$ into $2D$ representation inside the CPU, next, the program create all the variables needed to execute the program inside the device and copy the data from the host to the device, after that, the *control program* invokes some CUDA function (called kernels) to be executed in parallel inside the device, these functions are: *find Knn and build graph, compute the shortest path - step1, compute the shortest path - step2,*and *remove outliers,* returning the cost matrix C that contains the geodesic distance for all points. Finally the control program *finds the low dimensional* calling a function inside the CPU (with the cost matrix C as a parameter) using the CULA library that internally calls some kernels functions inside the device for calculation and the results are presented to the user.

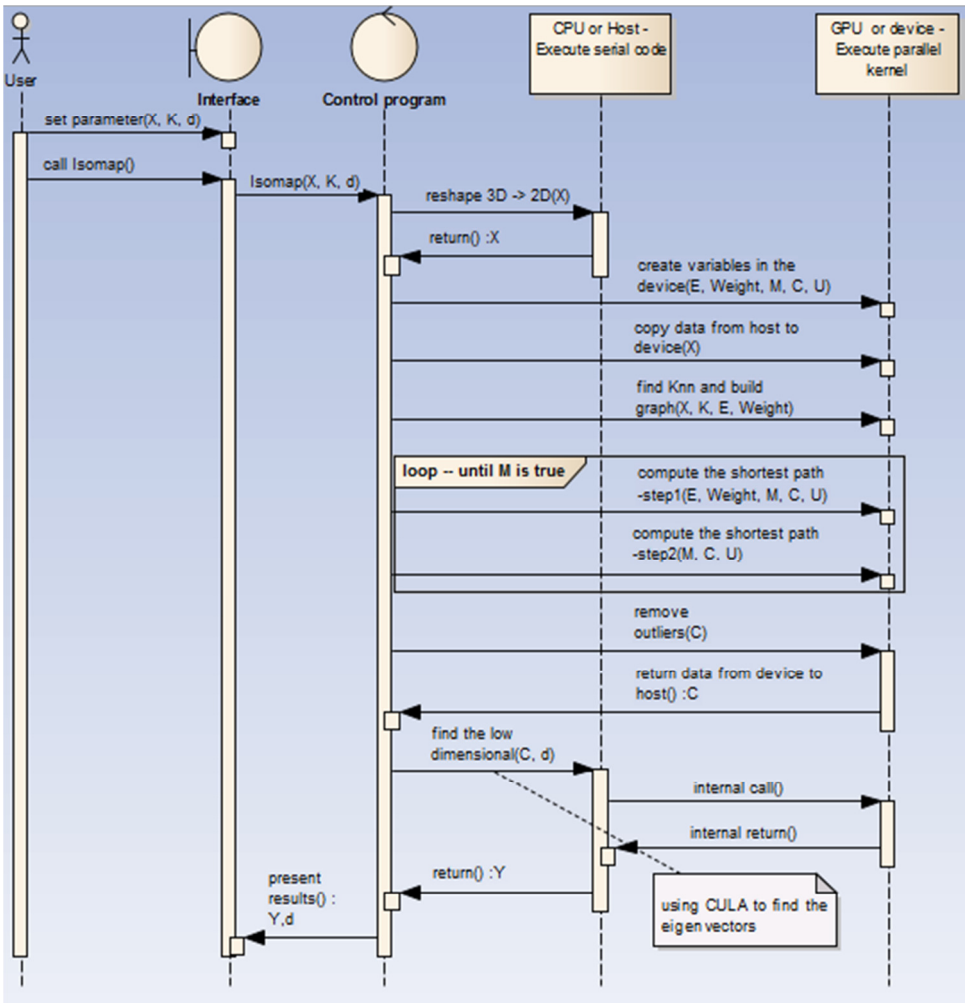

Figure 3-3: Process for Isomap

3.2 Locally linear embedding in CUDA

The naive implementation of locally linear embedding – LLE is described in Algorithm 5, for more details see Section 2.5.2. LLE algorithm has the same parameters as Isomap algorithm.

The LLE algorithm is show in Algorithm 5. Line 1 was also implemented in CUDA by (Garcia, Debreuve, & Barlaud, 2008) whose results are the matrices of distances and indexes of K nearest neighbors for each point x_i. Line 2 Compute the weights matrix w_{ij} that best linearly reconstructs x_i from its neighbors and minimizes the cost function in the same line. The pseudocode of line 2 is given in Algorithm 6. Finally, the line 3 of LLE algorithm was implemented using CULA, but is too slow when trying to compute the eigenvalues and eigenvectors for dense matrix.

Algorithm 5 LLE (X, K, d)

1. For each data point $x_i \in X$, find its K nearest neighbors.
2. Compute the weights matrix w_{ij} that best linearly reconstruct x_i from its neighbors and minimizes following cost function:

$$\varepsilon(w) = \sum_i \left| x_i - \sum_j w_{ij} x_j \right|^2 .$$

   The sum of the coefficients of w_{ij} must be equal to one: $\sum w_{ij} = 1$.
3. Find the d-dimensional embedding vectors y_i by using the weights w_{ij}, which minimizing the following cost function:

$$\varphi(Y) = \sum_i \left| y_i - \sum_j w_{ij} y_j \right|^2 .$$

A good solution in line 3, is to implement a Mex function in Matlab to call a function implemented in CUDA, that return the matrix M thought the computation of line 1 and 2, after that, convert the dense matrix M into a sparse matrix to find eigenvalues and eigenvectors in a faster way, needed to solve line 3. Algorithm 6 shows a CUDA

implementation to solve for reconstruction weights. This algorithm uses a data set X of size NxN and an edge Neighbors matrix that represent the neighbors or indices of size $N \times K$.

Algorithm 6 CUDA Solve for reconstruction weights (X, Neighbors)

1: **for all** points $i \leftarrow 1{:}N$ **do in parallel**
2: Create matrix Z consisting of K neighbors of x_i
3: Subtract x_i from every column of Z
4: Compute the local covariance $C \leftarrow Z' \times Z$
5: Regularization of C if (K > D)
6: Solve linear system $C\ x\ w \leftarrow 1$ for w
7: Enforce sum(w) ←1
8: Fill $W(1..K, i) \leftarrow w$;
9: **end**
10: **for all** coefficient in W of x_i $i \leftarrow 1{:}N$ **do in parallel**
11: Create matrix $M \leftarrow (I - W)' \times (I - W)$
12: **end**

In each iteration of Algorithm 6, each thread from line 2 to line 5 compute the local covariance C of size $K \times K$ (small matrix). That is, input to solve the linear system in line 6 $C \times w \leftarrow 1$, which was implemented in parallel using LU decomposition described in (Press, Flannery, Teukolsky, & Vetterling, 1992). Finally the result of line 11 is the matrix M.

3.2.1 Process for LLE

The complete process for LLE algorithm is shown in **Error! Reference source not found.**. The user through the interface, enter the parameters X, K, and d and call LLE for execution, the interface call to the *control program* who is responsible for executing the complete

program of LLE. The *control program* execute some parts of the program inside the CPU and another parts inside the GPU.

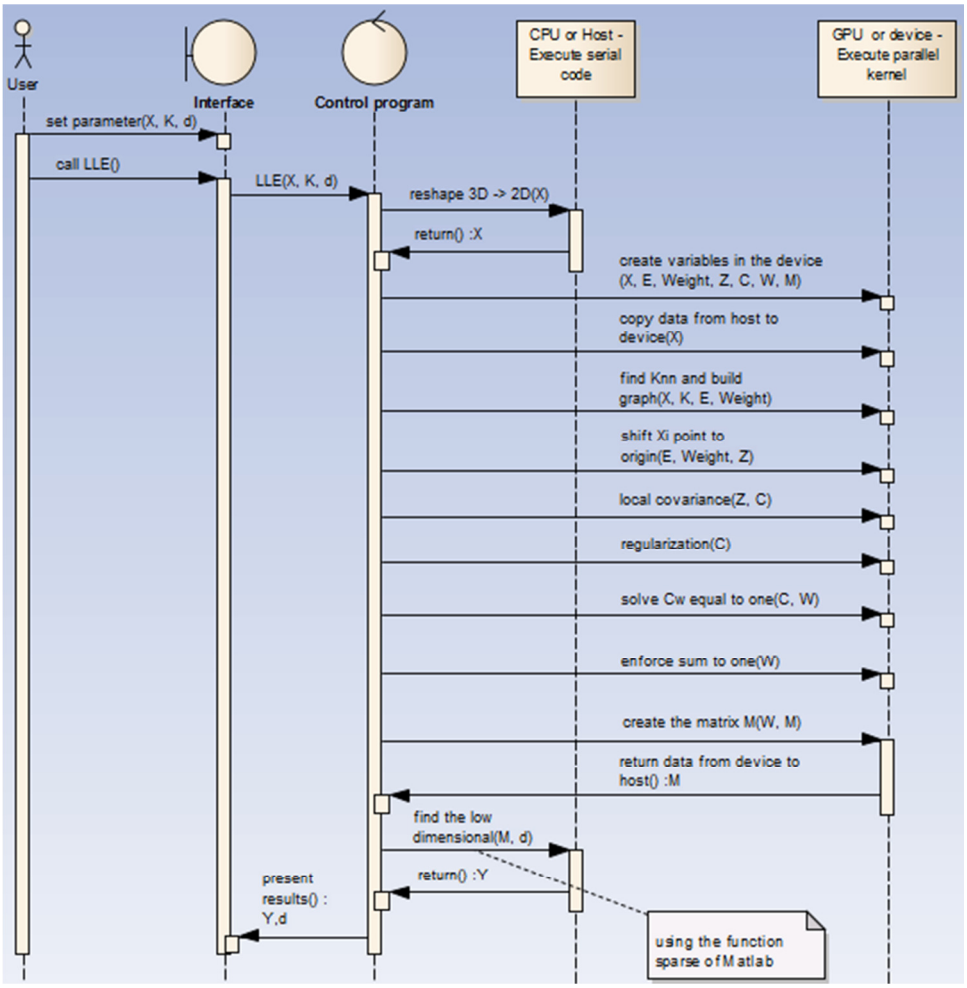

Figure 3-4: Process for LLE

First the program reshape the image from $3D$ into $2D$ representation inside the CPU, next, the program create all the variables needed to execute the program inside the device and copy the data from the host to the device, after that, the *control program* invokes some CUDA function (called kernels) to be executed in parallel inside the device, these functions are: *find*

Knn and build graph, shift x_i *point to origin, find the local covariance, make a regularization, solve* Cw *equal to one, enforce sum to one,* and *create the matrix* M, returning the matrix M that contains the embedding from eigenvectors. Finally the control program *finds the low dimensional* calling a function inside the CPU (with the embedding matrix M as a parameter) using the sparse function in Matlab, because is faster that the CULA library for dense matrix to find the eigenvectors and the results are presented to the user.

3.3 Laplacian eigenmap in CUDA

The Laplacian eigenmap algorithm is show in Algorithm 7, for more details see 0. Laplacian eigenmpas algorithm has the same parameters like Isomap algorithm and LLE.

Line 1 is the same as Algorithm 5, was implemented in CUDA by (Garcia, Debreuve, & Barlaud, 2008) whose results are the matrices of distances and indexes of K nearest neighbors for each point x_i. Line 2 assign the edge weights, which are determined using one of the second variation option b, where $\varepsilon = \infty$, no parameter is used. Where $W_{ij} = 1$, if vertices i and j are connected by an edge and $W_{ij} = 0$ otherwise. The pseudocode of line 2 is given in Algorithm 8. Finally, the line 3, of Laplacian eigenmap algorithm as LLE algorithm was implemented using CULA, but is too slow when trying to compute the eigenvalues and eigenvectors for dense matrix. A good solution in line 3 was to implement a Mex function in Matlab to call a function implemented in CUDA, that return the matrix D thought the

computation of line 1 and 2, after that, convert the dense matrix D into a sparse matrix to find eigenvalues and eigenvectors in a faster way, needed to solve line 3.

Algorithm 7 Laplacian eigenmap (X, K, d)

1. For each data point $x_i \in X$, find its K nearest neighbors.
2. Assign edge weights. The edge weights are determined using one of the following variations:
   a) Heat kernel uses a parameter $\varepsilon \in \mathbb{R}$. Especially if two nodes i and j are connected, put $W_{ij} = e^{-\frac{\|x_i - x_j\|^2}{\varepsilon}}$; otherwise, $W_{ij} = 0$.
   b) Simple form is getting if $\varepsilon = \infty$ in $W_{ij} = e^{-\frac{\|x_i - x_j\|^2}{\varepsilon}}$, no parameter is used. Where $W_{ij} = 1$, if vertices i and j are connected by an edge and $W_{ij} = 0$ otherwise. This simplification avoids the need to choose ε.
3. Compute Laplacian eigenmap, solving the generalized eigenvalue problem $LY = \lambda DY$, where D is diagonal weight matrix, and its entries are column (or row, since W is symmetric) sums of W, $D_{ii} = \sum_j W_{ij}$. $L = D - W$, is the Laplacian symmetric, positive semidefinite matrix.

Algorithm 8 shows a CUDA implementation to assign the edge weights according to the neighbors of x_i. This algorithm is similar to LLE algorithm.

Algorithm 8 CUDA Assign edge weights (X, Neighbors)

1: **for all** points $i \leftarrow 1{:}N$ **do in parallel**
2: **if** $x_j \in Neighbors_{x_i}$ **then**
3: Set $W_{ij} = 1$ and $W_{ji} = 1$
4: **else**
5: Set $W_{ij} = 0$ and Set $W_{ji} = 0$
6: **end**
7: **end**
9: **for all** points $i \leftarrow 1{:}N$ **do in parallel**
11: $D \leftarrow$ Count in $W_{row \leftarrow i}$ where $W_{row,j:1..N} = 1$
12: Solve $D \leftarrow D - W$
13: **end**

3.3.1 Process for Laplacian eigenmap

The complete process for Laplacian eigenmap algorithm is shown in Figure 3-5. The user through the interface, enter the parameters X, K, and d and call Laplacian eigenmap for execution, the interface call to the *control program* who is responsible for executing the complete program of Laplacian eigenmap. The *control program* execute some parts of the program inside the CPU and another parts inside the GPU. First the program reshape the image from $3D$ into $2D$ representation inside the CPU, next, the program create all the variables needed to execute the program inside the device and copy the data from the host to the device, after that, the *control program* invokes some CUDA function (called kernels) to be executed in parallel inside the device, these functions are: *find Knn and build graph,* assign edge weights, and counting neighbors, returning the matrix D. Finally the control program *finds the low dimensional* calling a function inside the CPU (with the matrix D as a parameter) using the sparse function in Matlab, because is faster that the CULA library for dense matrix to find the eigenvectors and the results are presented to the user.

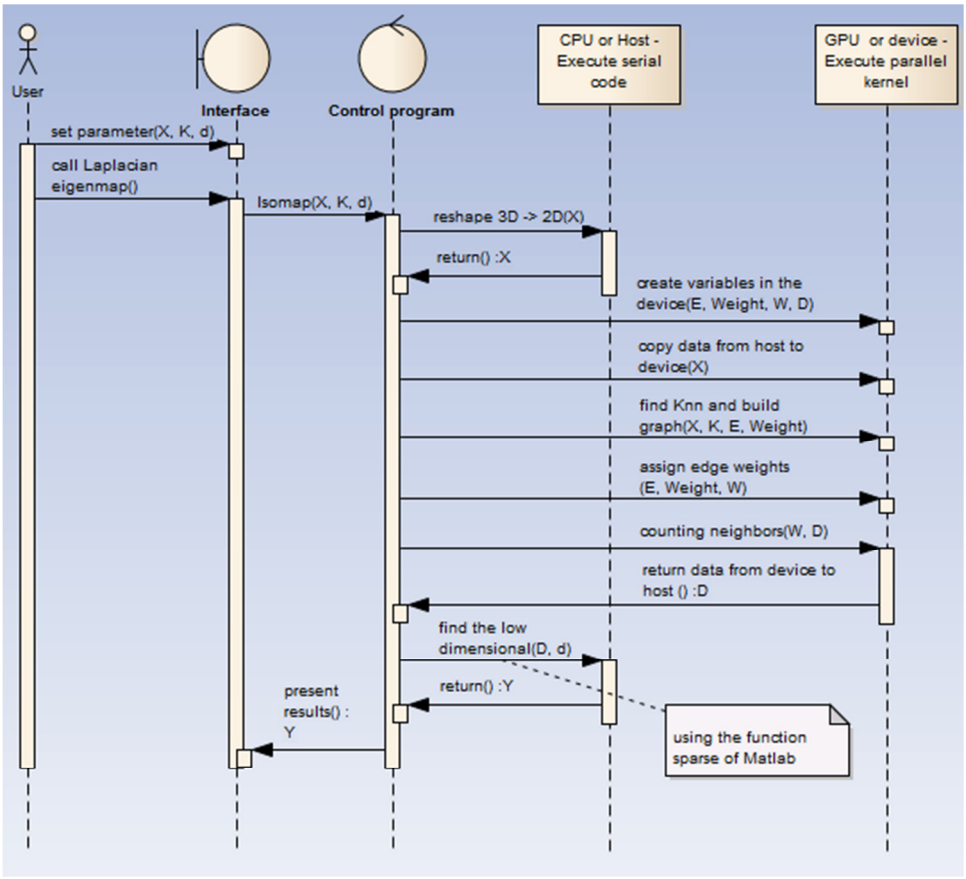

Figure 3-5: Process for Laplacian eigenmap

4 EXPERIMENTAL RESULTS

All CUDA experiments were conducted on a 64-bit workstation equipped with a quad-core Intel® Xeon with 12 GB RAM and two NVIDIA Tesla C1060 GPU cards with total dedicated memory of 4 GB each one. Tesla C1060 is based on the massively parallel, many-core Tesla processor, which is coupled with the standard CUDA C programming environment to simplify many-core programming.

Figure 4-1 : Tesla C1060 computing processor board (Zone, 2011)

Table 4-1: Tesla Board Configuration

# of Tesla GPUs	1
# of Streaming Processor Cores	240
Frequency of processor cores	1.3 GHz
Single Precision floating point performance (peak)	933
Double Precision floating point performance (peak)	78
Floating Point Precision	IEEE 754 single & double
Total Dedicated Memory	4 GDDR3
Memory Speed	800MHz
Memory Interface	512-bit
Memory Bandwidth	102 GB/sec
Software Development Tools	C-based CUDA

With the operative system running Windows 7, the software installed is the driver for NVIDIA Tesla C1060 GPU card which supports the version of CUDA 3.2, an environment to compile and edit 64-bit CUDA applications, because CUDA compiler has a dependency with C++ compiler. This environment is supported by Visual Studio 2008 C++, which in combination with CUDA, allows developers to leverage the CPU for parallel tasks and the GPU for massively parallel computing, and finally Matlab was used to present all the result.

4.1 Testing of the algorithms

Indian Pines - Hyperspectral Image

Indian Pines is provided by Laboratory of Applied Remote Sensing (LARS) at Purdue University (University, 2010). The image size is 128 by 128 pixels with 200 bands. The data intensities was normalized in the [0,1] range.

(a) **(b)**

Figure 4-2: a) Indian Pines image (RGB shows bands 29, 20, 11), b) Ground truth.

The image was taken in 1992, covering the NW Indiana's Indian Pines Site 3, an agriculture area. Figure 4-2 (a) shows the image for bands (RGB 29, 20, and 11). The ground truth here is available as an information image. There are 16 land cover classes (see Figure 4-2 (b)).

Enrique Reef - Hyperspectral Image

Enrique Reef, Puerto Rico: A high spatial-spectral resolution image of Enrique Reef Puerto Rico area has been collected using the NASA HYPERION sensor, which is located onboard the Earth Observing-1 satellite. This image contains 124 by 128 pixels and 204 bands. Enrique Reef data set includes ocean, reef, sand, mangrove, and sea grass habitats. The data intensities was normalized in the [0,1] range. Figure 4-3 shows the image for bands (RGB 32, 21, and 14).

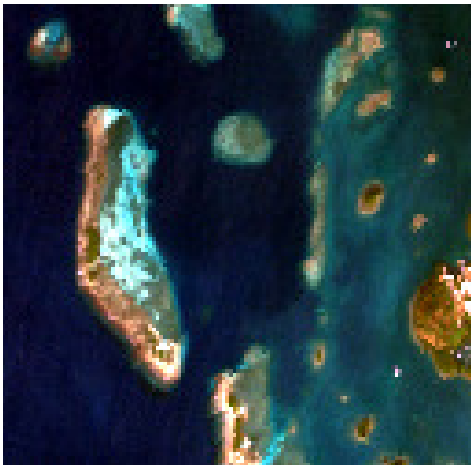

Figure 4-3: Enrique Reef image (RGB shows bands 32, 21, 14)

Fish Boat - Hyperspectral Image

The image size is 128 by 128 pixels with 60 bands. The data intensities was normalized in the [0,1] range. Figure 4-4 shows the image for bands (RGB 38, 47, and 56).

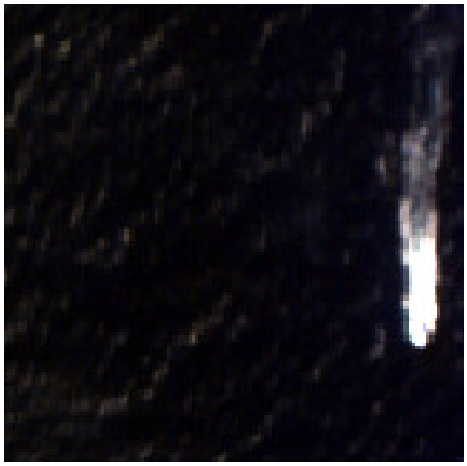

Figure 4-4: Enrique Reef image (RGB shows bands 38, 47, 56)

The nonlinear dimensionality reduction implementation in CUDA has been tested using the three hyperspectral images previously described. The results are summarized in Figure 4-5, Figure 4-6, and Figure 4-7. The intrinsic dimensionality considered for visualization was $d = 3$, and the number of the nearest neighbors are K = 7, 12, 17, and 22. The value of this parameter affects the outcome.

The algorithm Isomap produces better results thought increasing the number of neighbors, but requires a huge amount of memory space to compute the geodesic distance and is very slow. In the case of LLE algorithm, produce good results similar to Isomap working with too much less information. This algorithm has to find a optimal solutions for each data point

using its neighbors, but is very sensible to the data and in some cases is not possible to find the solution, as shown in Figure 4-6 and Figure 4-7. Now Laplacian eigenmap is very similar to LLE because work with the information of its neighbors for each data point and create graph with the entries of each neighbor's point.

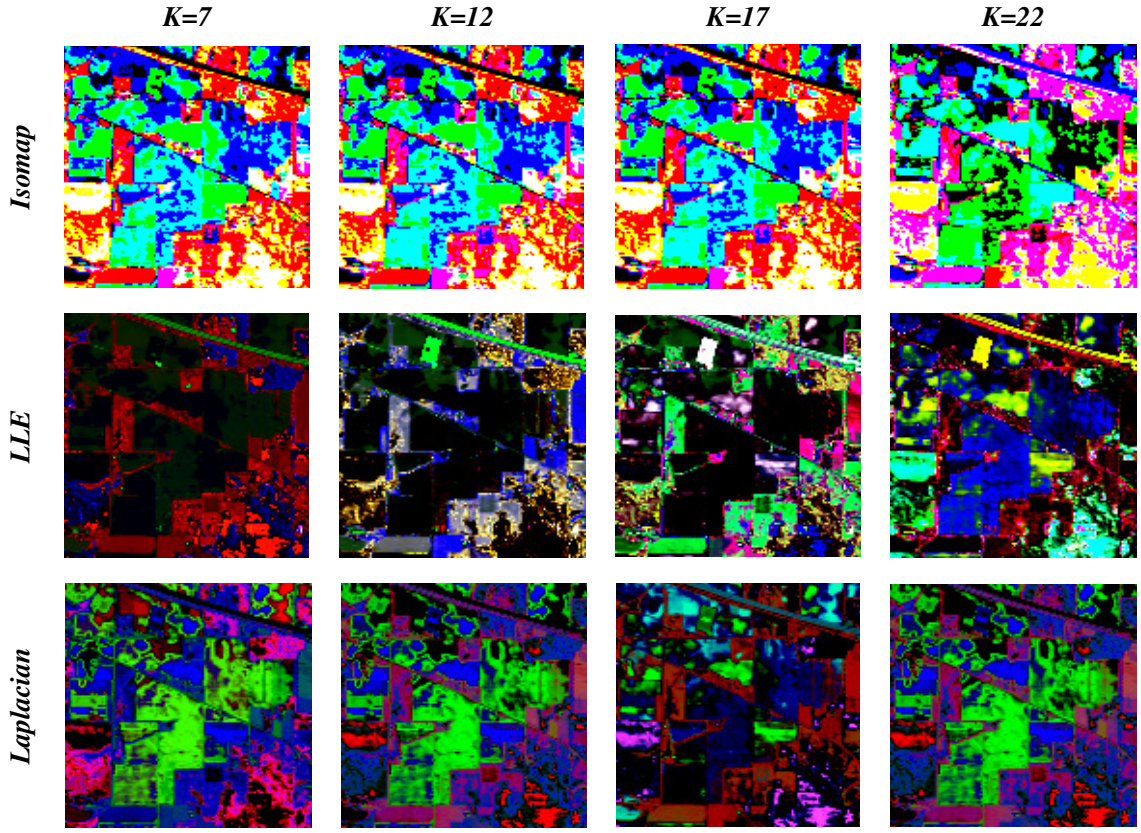

Figure 4-5: Manifold learning applied to Indian Pines

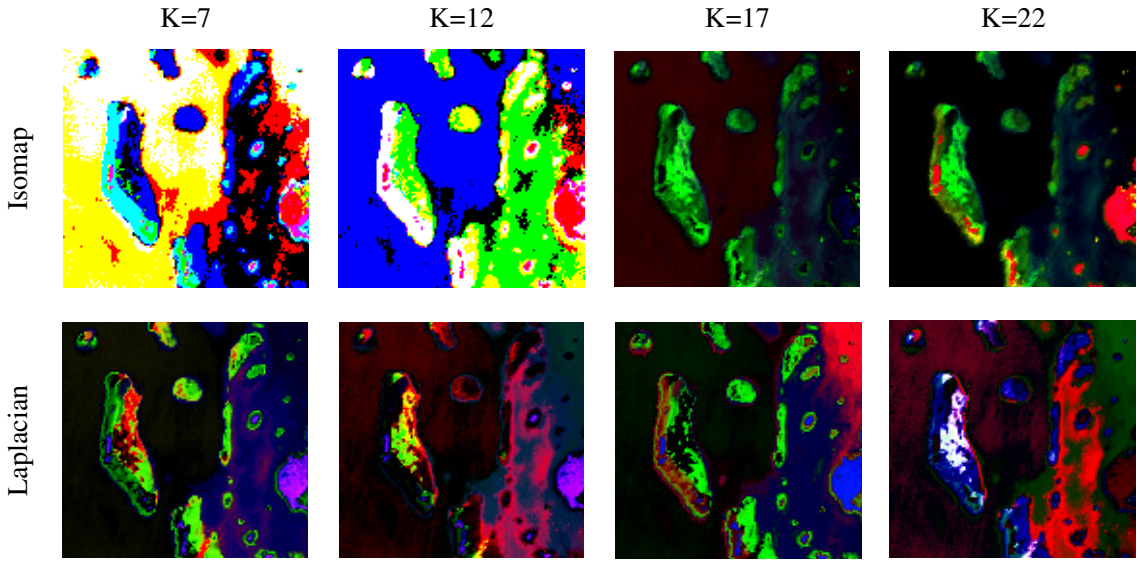

Figure 4-6: Manifold learning applied to Enrique Reef

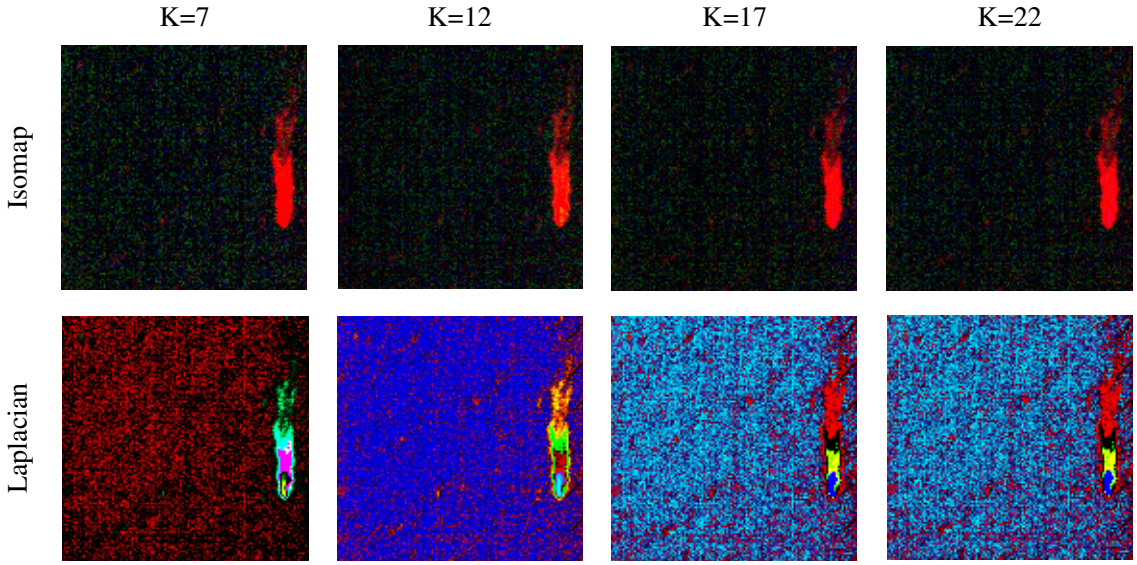

Figure 4-7: Manifold learning applied to Fish Boat

4.2 Speedup between pure C++ and CUDA implementations

To measure the speedup (S_p) between two different implementation was used the Equation 4.1.

$$S_p = \frac{T_1}{T_p} \quad \textbf{4.1}$$

Where:

p is the number of processors

T_1 is the execution time of the sequential algorithm

T_p is the execution time of the parallel algorithm with p processors.

In order to compare the running time of CUDA implementation of manifold learning algorithm was necessary to develop the same program in pure C++. Additional libraries were used, like Intel® Math Kernel Library (Intel® MKL) (Intel®, 2011) to solve eigenvectors and eigenvalues, and the library OpenMP C++ (Board, 2010) application program interface to run the parallelizable parts of pure C++ implementation using multiple processors. However just the implementation part in pure C++ and CUDA that we developed, was compared to measure the speedup between this implementation. The following figures show the speedup of the implementation of Isomap, Locally linear embedding and, Laplacian eigenmap.

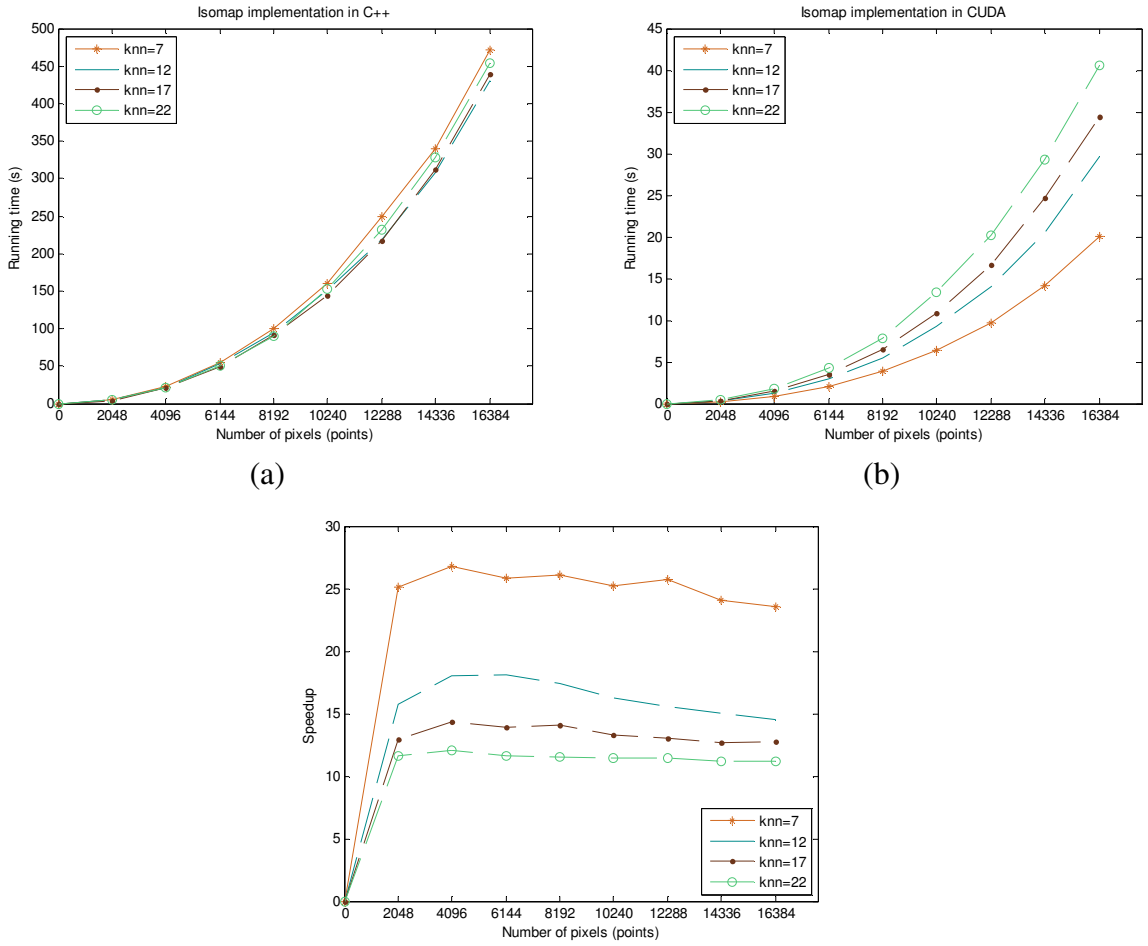

Figure 4-8: Isomap implementation - (a) Using pure C++, (b) Using CUDA, and (c) Speedup between (a) and (b)

In Figure 4-8 (c) the speedup of CUDA part of Isomap, is 26.81 times faster that its CPU counterpart for $K = 7$. In the case of locally linear embedding, show in Figure 4-9 (c), the speedup is 18.46 for $K = 7$, and Laplacian eigenmap, show in Figure 4-10 (c), has a speedup of 16.32 for $K = 12$. In all of these algorithms as the number of K nearest neighbors increase, the value of speedup decrease. Isomap has a lot time consuming to find the geodesic distance using Dijkstra's algorithm.

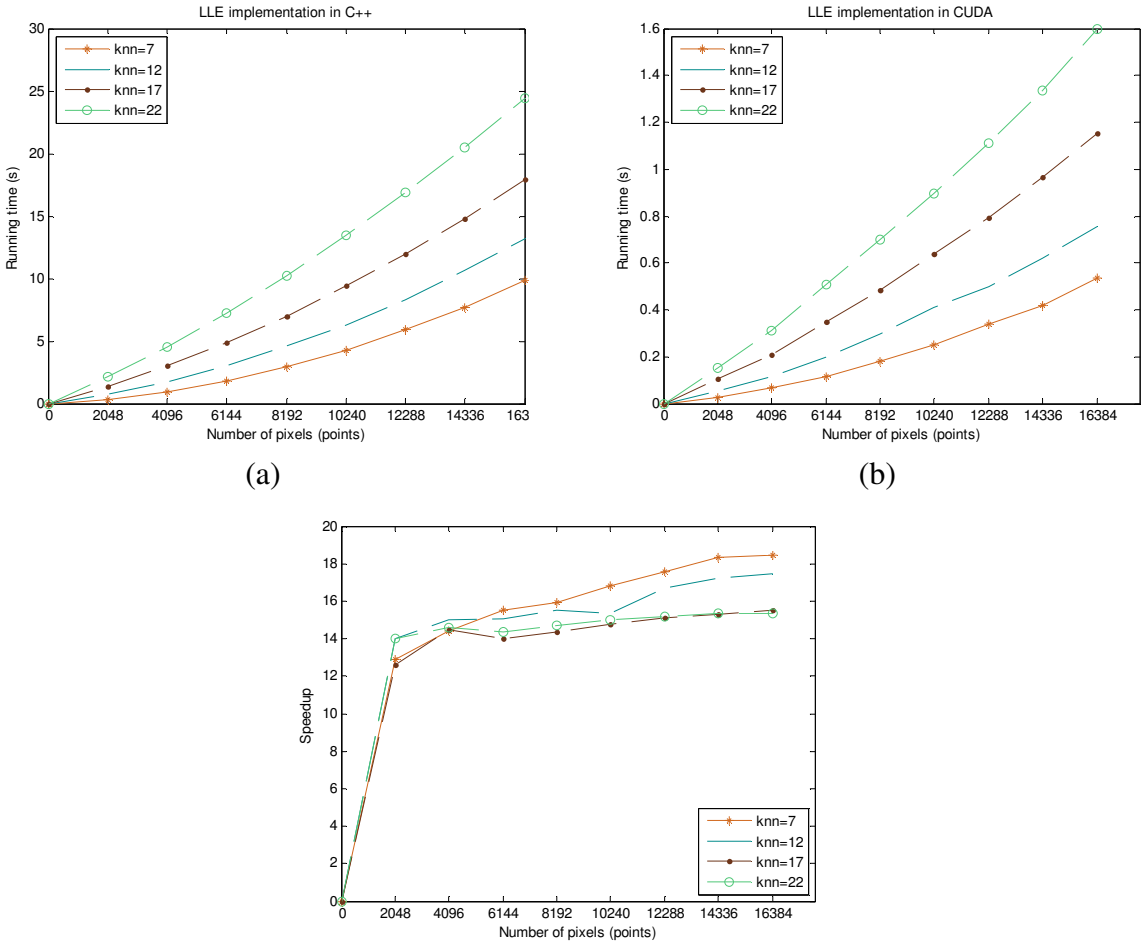

Figure 4-9: LLE implementation - (a) Using pure C++, (b) Using CUDA, and (c) Speedup between (a) and (b).

In the case of LLE, see Figure 4-9, it is very sensible to data, because for each data we must find the coefficient of linear system solution, but in some cases it does not have a solution, and therefore it is not possible to build a graph for this point. The last algorithm Laplacian eigenmap, see Figure 4-10, is the easiest algorithm that is not sensible to the data and is faster that the other methods, but loses more information.

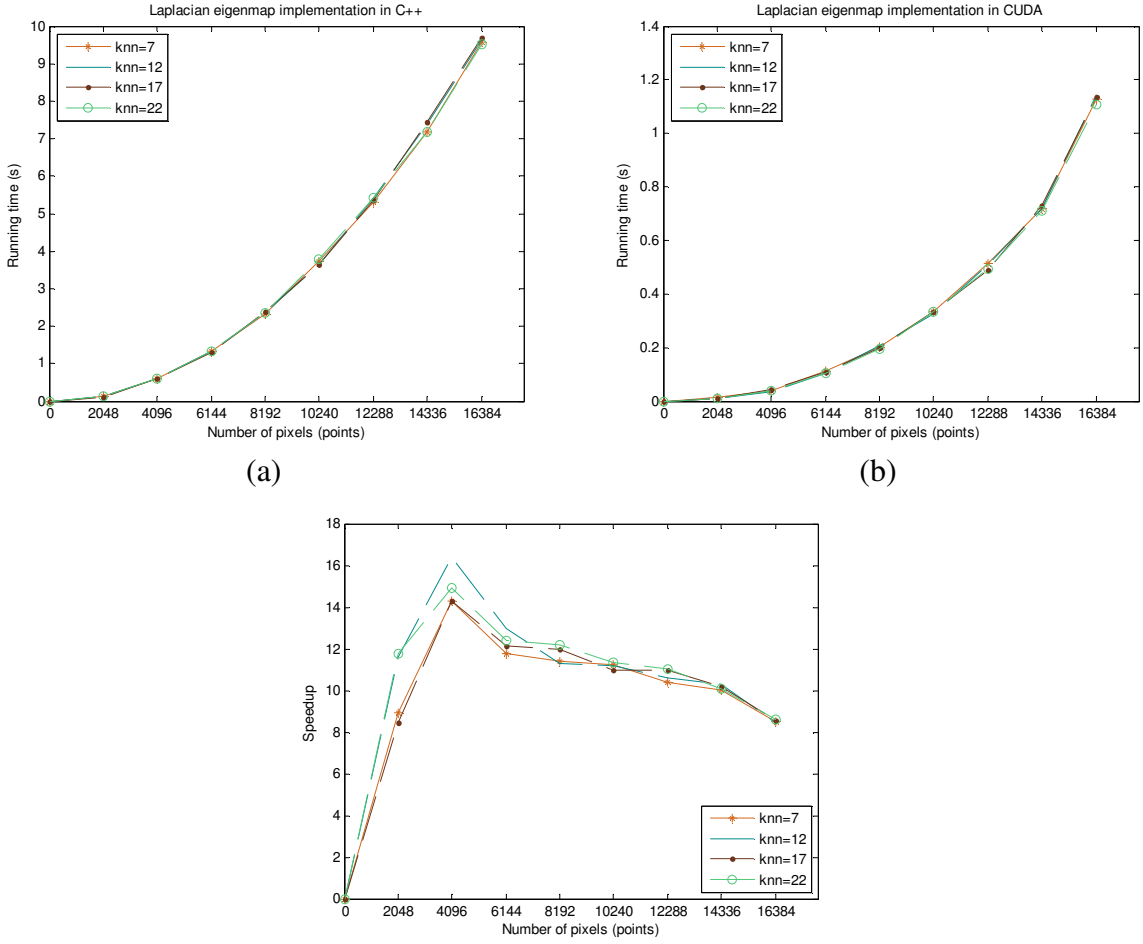

Figure 4-10: Laplacian eigenmap implementation - (a) Using pure C++, (b) Using CUDA, and (c) Speedup between (a) and (b).

5 CONCLUSIONS AND FUTURE WORK

Conclusions

In this report, we presented CUDA implementations of three most famous manifold learning algorithms for real hyperspectral images. These algorithms have wide practical applications like visualization, segmentation, and classification. We presented fast solution for Isomap, locally linear embedding, and Laplacian eigenmap, whose CUDA part implementation runs 26.81, 18.46, and 16.38 times faster respectively, than pure C++ implementation for matrices with order 2048 to 16384. The NVIDIA® Tesla™ C1060 transforms a workstation into a high-performance computer that outperforms a small cluster. Experiments showed good scalability on data sets.

All our dimension reduction algorithms basically consisted of three general steps, the first step is to find the K nearest neighbors, while in step 2, it constructs a graph of distances (or weights), which will be used in step 3 to find the desired d low dimension representation.

Our work consisted in implementing the step 2 in CUDA, because steps 1 and 3 were previously implemented in CUDA. The first one was developed by (Pawan & Narayanan, 2007) and the third step was implemented using the optimized CULA library.

An important detail is that Isomap working with dense matrices, while LLE and Laplacian eigenmap work with sparse matrices, therefore the step 3 of Isomap algorithm was implemented using the CULA library because this library works only for dense matrices,

while for sparse matrices was necessary to use the functions of Matlab to find the eigenvectors since it is much faster than the CULA implementation.

Another important thing was to develop a full version of the algorithm in pure C++ whose step 3 was implemented using the MKL LAPACK library to measure the speedup between this and the CUDA implementation. In this work we presented only the speedup of the second steps of the algorithms.

Future Work

The future work to this research will be to implement another variant of Isomap that avoid building the geodesic distance for all points, instead a subset of points. In the case of LLE and Laplacian eigenmap there is not a CUDA implementation at this moment to compute the eigenvectors and eigenvalues for sparse matrix.

References

Belkin, M., & Niyogi, P. (2003). Laplacian eigenmpas for dimensionality reduction and data representation. *Neural computation*, (pp. 16(5):1373-1396).

Board, T. O. (2010, July). Retrieved from The OpenMP API specification for parallel programming: http://openmp.org/wp/

Center, s. o. (2011, April). *Remote sensing*. Retrieved from http://www.scioly.org/wiki/Remote_Sensing

Garcia, V., Debreuve, E., & Barlaud, M. (2008). Fast k nearest neighbor search using GPU. *Computer vision and pattern recognition workshops, 2008. CVPRW '08. IEEE computer society conference on*, (pp. 1-6, 23-28).

Geiger, A., Urtasun, R., & Darrell, T. (2009, June). Rank priors for continuous non-linear dimensionality reduction. *Computer vision and pattern recognition, 2009. CVPR 2009. IEEE conference on*, 880-887.

Intel®. (2011). Retrieved from Intel® Math Kernel Library (Intel® MKL) : http://software.intel.com/en-us/articles/intel-mkl/

Ivancevic, V., & Ivancevic, T. (2007). *Applied differential geometry: a modern introduction.* Singapore, Singapore: World scientific publishing.

Jianru, X., & Nanning, Z. (2009). Manifold related terminologies. In *Statistical learning and pattern analysis for image and video processing (advances in pattern recognition)* (pp. 327-332). London: Springer; 2nd printing. edition (August 19, 2009).

Lee, J., & Verleysen, M. (2007). *Nonlinear dimensionality reduction (information science and statistics)* (1st Edition ed.). New York, NY, USA: Springer Science + Business Media, LLC.

Pawan, H., & Narayanan, P. (2007). Accelerating large graph algorithms on the GPU using CUDA. *High performance computing – HiPC 2007 in high performance computing – HiPC 2007, Vol. 4873* (pp. 197-208). Germany: Springer.

Photonics, E. (2010). Retrieved from CULA tools: http://www.culatools.com/

Press, W., Flannery, B., Teukolsky, S., & Vetterling, W. (1992). LU decomposition and its applications. In *Numerical recipes in C book set: Numerical recipes in C: The art of scientific computing* (pp. 43-48). Cambridge University Press; 2 edition.

Roweis, S., & Saul, L. (2000). Nonlinear dimensionality reduction by locally linear embedding. (pp. 2323--2326). Science.

Schott, J. R. (2007). *Remote sensing -- the image chain approach.* New York: Oxford University Press.

Shaw, G., & Burke, H. (2003). Spectral imaging for remote sensing. *Lincoln Lab- oratory Journal*, Vol. 14, no.1.

Talwalkar, A., Kumar, S., & Rowley, H. (2008, June). Large-scale manifold learning. *Computer vision and pattern recognition, 2008. CVPR 2008. IEEE Conference on*, 1-8, 23-28.

Tenenbaum, J., de Silva, V., & Langford, J. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, (pp. vol. 290, no. 22, pp. 2319-2323).

University, l. o. (2010, April). Retrieved from Hyperspectral images: https://engineering.purdue.edu/~biehl/MultiSpec/hyperspectral.html

Zone, N. D. (2011, January). Retrieved from NVIDIA CUDA C programming guide: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf