Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Mohamed Ismail

HW/SW Co-design Implementation of Hyperspectral Image Classification Algorithm

Master's thesis in Embedded Computing Systems
Supervisor: Milica Orlandic
June 2020

HYPSO
· NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY · NTNU SMALLSAT ·

NTNU
Norwegian University of
Science and Technology

Mohamed Ismail

HW/SW Co-design Implementation of Hyperspectral Image Classification Algorithm

Master’s thesis in Embedded Computing Systems
Supervisor: Milica Orlandic
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Abstract

Hyperspectral Imager for Oceanographic Applications (HYPSO) mission is being developed as a part of SmallSat laboratory at NTNU. The goal of the mission is to develop, launch, and operate a series of small satellites specially made for maritime observation and surveillance, e.g. monitoring of algae, plankton, oil spills, arctic ice etc. Satellites will be equipped with Zynq-7000 on-board processing system consisting of ARM®-based processor with the hardware programmability of an FPGA.

In this thesis, on-board hyperspectral image classification on FPGA is explored. Based on the hyperspectral data collected by the satellites, the detection of algae blooms or other material in the Norwegian sea can processed by classification. Classification aims to distinguish and label material signatures on a given hyperspectral image. This thesis presents implementations of clustering-type unsupervised classification algorithms for hyperspectral images. Two algorithms are implemented in software; a novel segment-based clustering algorithm and a spectral clustering algorithm. The software implementations are implemented in MATLAB R2019a and Python 3.7.3, respectively. Expirements on the algorithms are carried on 8 different publicly available hyperspectral scenes with ground truth data and results are compared with a state-of-the-art segment-based clustering method using NMI and Purity evaluation scores. Results of the software experiments are further analysed prior to FPGA implementation.

Based on the software analysis, spectral clustering algorithm is implemented as a novel hardware-software partitioned system on Xilinx Zynq-7000 development platform. The hardware-software co-design is implemented to gain the most efficiency compared to the standalone software solution. FPGA VHDL modules for graph construction are developed and synthesized using Xilinx Vivado Design Suite 2019.1 and HDL Coder™ R2020a from MathWorks. Eigenvalue and eigenvector decomposition is implemented using Vivado HLS 2019.1 and synthesized using Xilinx Vivado tool. This implementation illustrates productivity benefits of a C-based development flow using high-level synthesis (HLS) optimization methods. Resource utilization, performance, and classification scores are reported for the HW/SW co-design of spectral clustering algorithm.

Preface

This master's thesis is the final part of my European Master degree in Embedded Computing Systems (EMECS). The thesis work was conducted at the Norwegian University of Science and Technology (NTNU) within a SmallSat project (HYPSO) and is a continuation of the work done in a specialization project at NTNU. Hyperspectral image classification on FPGAs is a challenging topic and represents a very promising area of research. This thesis allowed me to both acquire and enrich my knowledge and skills by exploring a wide spectrum of topics and tools involved in hyperspectral image classsification. Moreover, as a student, I find it always wonderful and exciting to learn new topics, and even though it has been a challenging learning experience, it has also been a rewarding and gratifying experience.

First of all, I would to like express my thanks, appreciation, and gratefulness to my supervisor Milica Orlandić, for guiding me through this work, for always being available for my questions and discussions, for enlightening me to possible research dimensions in my work, and for emotionally and academically supporting me through the events of the 2020 coronavirus global pandemic.

Finally, a special thanks to my family and friends for being a very solid support system, a positive company, and for ensuring I am physically and psychologically well through these months and weeks.

Mohamed Ismail
June 24, 2020.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

1.1 Motivation

Under certain conditions in the sea, phytoplankton (tiny microscopic plants) may grow out of control and form harmful algal blooms (HABs). A bloom does not have to produce toxins in order to be harmful to the environment. It can also cause anoxic conditions, where oxygen is depleted from the water. Dense blooms can block light to organisms lower in the water column, or even damage fish gills. On the other hand, blooms may also produce extremely toxic compounds that have a detrimental effect on fish, shellfish, mammals, birds, and even people [1]. Harmful algal blooms are one of the biggest challenges to fish health and welfare in Norway. In 2019, an algae bloom killed most of the fish in the Norwegian sea and as a result, incurred a loss of 2.4 billion Norwegian Kroner (NOK) to the salmon farming industries in Norway as reported by the Norwegian Seafood Research Fund (FHF) [2]. This is where HYPSO mission makes its impact by trying to find indicators of water pollution, algae bloom and other metrics that may inflict the well being and quality of the fish.

Hyper-Spectral Imager for Oceanographic Applications (HYPSO) mission is being developed at the NTNU SmallSat lab, Trondheim [3]. The main focus of the mission is for both oceanographic measurements and synoptic in-situ field measurements. This is a novel approach and has a remarkable potential for reducing cost and improving data quality in oceanography. The imager will observe the oceanographic phenomena by using a small satellite equipped with a hyperspectral camera on-board, operating in cooperation with aerial, surface, and underwater vehicles [3].
Based on the hyperspectral data that is collected, the detection of algae blooms can done by classification. Luckily algae blooms are very distinct in the spectral domain, which is part of the data obtained by the HSI. This way, with a good enough classification algorithm, one is able to detect algae blooms and classify whether it is a harmful bloom or not based on their spectral signatures. This master thesis is a part of the HYPSO mission.

1.2 HSI Classification in the context of HYPSO mission

Hyperspectral image classification is the task of assigning a class label to every pixel on the image. In the context of HYPSO mission, HSI classification should provide fast, meaningful, and accurate characterisation to the obtained hyperspectral image. Some of the objectives of the mission are listed below [3]:

- To provide and support ocean color mapping through a Hyperspectral Imager (HSI) payload, autonomously processed data, and on-demand autonomous communications in a concert of robotic agents at the Norwegian coast;
- To collect ocean color data and to detect and characterize spatial extent of algal blooms, measure primary productivity using emittance from fluorescence-generating micro-organisms, and other substances resulting from aquatic habitats and pollution to support environmental monitoring, climate research and marine resource management;
- Operational data shall be compressed, have at least 20 spectral bands, and include radiometric calibration, atmospheric correction, classification, super-resolution and target detection;

Classification of hyperspectral images has been initiated since 1980 by using traditional multispectral classification approaches. Most widely used classifiers are known as "hard or traditional classifiers" [4]. However, such methods face challenges dealing with high resolution data and produce inaccurate classification results. Significant efforts have been reported for classification and feature extraction of hyperspectral images methods. Such advancements reported high accuracies and good efficiency on the state-of-the-art public datasets. These classification algorithms can be divided into two categories:

1. Supervised learning HSI classification.
2. Unsupervised learning HSI classification.

The supervised classification is the process of using samples of known identity to assign unclassified pixels to one of several informational classes. On the other hand, the unsupervised one is the identification of natural groups or structures on a given image dataset without the user providing sample classes.

In the literature, existing hyperspectral image classification algorithms, both supervised and unsupervised, may suffer from "curse of dimensionality" resulting from large number of spectral dimensions, high computational complexity, and scarcity of labelled training examples [5]. The specialization project work provided detailed analysis on the state-of-the-art for supervised and unsupervised classification methods [6]. Based on the analysis, unsupervised classification development is chosen for the thesis. An improved shorter summary version of the specialization project analysis is in section 2.3.
In this thesis, two machine learning algorithms based on unsupervised clustering classification are developed and implemented in software. The two algorithms are experimented with eight publicly available HSI datasets and their results are compared. Based on the result analysis, hardware-software co-design for FPGA platform is developed. The algorithms do not require prior knowledge about the hyperspectral images and hence, are applicable in the context of the HYPSO mission where there is a scarcity of water based hyperspectral images datasets.

1.3 HYPSO mission payload

At NTNU, the HYPSO team is working to design, develop, and test a hyperspectral payload for a small satellite, called HYPSO satellite. The satellite will carry a hyperspectral payload to provide hyperspectral images to the Autonomous Ocean Sampling Network (AOSN). As depicted in Figure 1.1, this network includes aerial, surface, and underwater vehicles working together to investigate the areas of interest using the data provided by the small satellite. The payload of the small satellite is to be integrated into a CubeSat framework delivered by Nano Avionics (NA) and is equipped with PicoZed board System-on-module (SoM) with a Zynq-7030 Processing System (PS). For testing purposes, Xilinx Zynq-7000 All Programmable System on Chip (SoC) is provided to test and process the hyperspectral data.

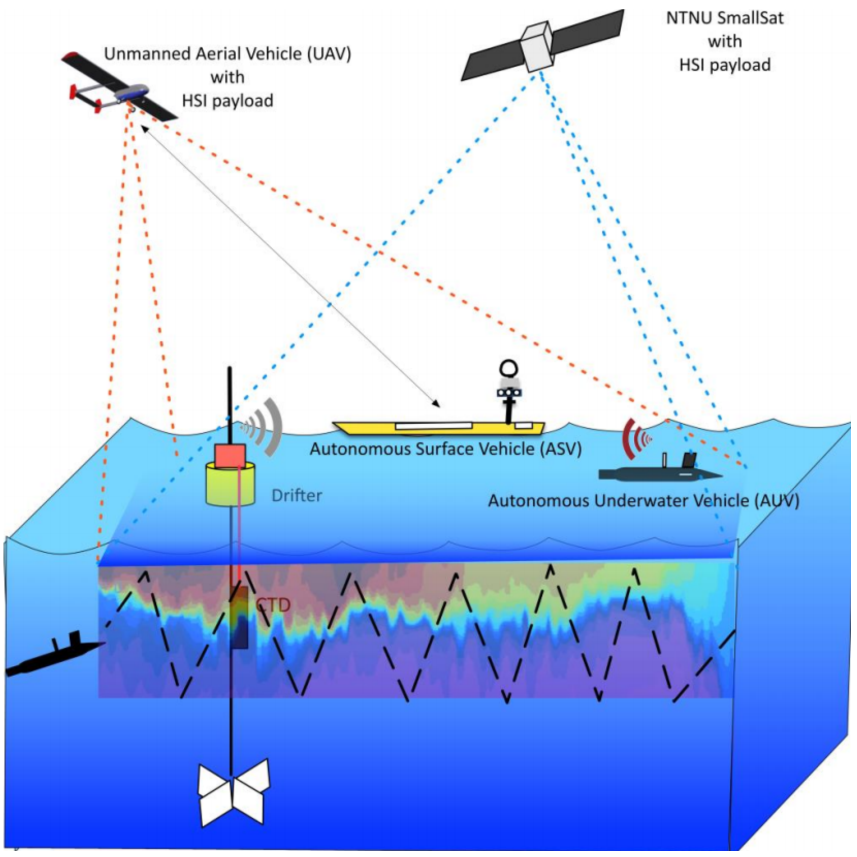

NTNU SmallSat
with
HSI payload
Unmanned Aerial Vehicle (UAV)
with
HSI payload
Autonomous Surface Vehicle (ASV)
Drifter
Autonomous Underwater Vehicle (AUV)
CTD

Figure 1.1: The Autonomous Ocean Sampling Network (AOSN) [3].

Owing to the complexity and dimensionality of HSI, many classification algorithms are considered computationally intensive. This often leads to the requirement of hardware accelerators to speed up computations. Over the years, FPGAs have become one of the preferred choices for fast processing of hyperspectral data [7]. FPGAs have significant advantages in efficiently processing HSIs due to the following reasons. First, FPGAs have competent levels of performance closing to those offered by GPUs with much lower power consumption. Another is that FPGA devices with increased levels of tolerance to ionizing radiation, making it suitable for space and are widely used as the solution for onboard processing at Earth observation satellites. In addition, FPGAs have the inherent ability to change their functionality through partial or full reconfiguration, hence, opens the possibility to select algorithms from a ground station [7].

The provided Xilinx Zynq-7000 SoC contains an ARM Cortex-A9 processor along with FPGA programmable logic and offers high performance ports to connect ARM processing system and FPGA programmable logic. This is very desirable for hardware and software codesigned implementations. By deciding which elements will be performed by the programmable logic, and which elements will run on the ARM Cortex-A9, the computation system is partitioned leading to the accelerated execution of hyperspectral processing algorithms including classification algorithms.

1.4 Main Contributions

The specialization project [6] recommended the fulfillment of the following tasks in the master thesis:

- Incorporation of methods like segmentation before classification (segment-based classification) for initially partitioning the hyperspectral image spatially. The proposed approach in the specialization project partitioned the regions by dividing the HSI image into k regions iteratively by going through the whole image; which is considered simple partitioning.
- Development of a two-level filtering algorithm before/during k-means clustering.
- Change tree structure. The kd-trees discussed in the project represent strict rectangular partitions of the image which is very useful for plotted data points but may not be suitable for hyperspectral images.
- Incorporation of dimensionality reduction methods including principal component analysis (PCA).

These tasks were approached in this thesis by development of two proposed solutions. The first one is a segment-based clustering where a full segmentation method based on Binary Partition Trees (BPT) processes the HSI data before filtering k-means clustering classification is applied. This framework represents a further development and enhancement to the method presented in the specialization project [6], making it a novel framework in literature. The second one is an adaptation of a near-real time clustering method for unsupervised HSI classification based on spectral clustering.
Both frameworks are developed in software and are experimented on eight different HSI datasets, including both land-cover and water-based datasets. Furthermore, experimental results are compared to state-of-the-art clustering methods. According to the analysis of the experiments and the implementation of the methods, a novel HW/SW co-design solution is developed for the chosen method: spectral clustering.

To the best of the author's knowledge, the HW/SW implementation in this work is a novel implementation for spectral clustering type methods on FPGA platforms. Consequently, a novel method for hyperspectral image clustering classification on FPGAs. In literature, the only clustering implementation for hyperspectral images on FPGAs has

been developed by [8] and is based solely on k-means clustering.
In addition, this thesis proposes a new segment-clustering software framework for hyperspectral images based on a three-stage scheme: (i) pre-segmentation, (ii) segmentation, and (iii) PCA and k-means clustering.

1.5 Structure of the Thesis

This section describes the organization of the rest of this thesis.

Chapter 2 discusses the background information necessary for development of the two HSI classification methods, for both software and hardware implementation. In addition, brief overview of Zynq-7000 platform is provided, with a short discussion on the state-of-the-art HSI classification algorithms.

Chapter 3 presents the software implementation of both methods.

Chapter 4 provides the experimental details including parameter settings. It also analyzes the results obtained by the experiments and accordingly, the bases of choosing a suitable algorithm for HW/SW implementation.

Chapter 5 discusses the HW/SW co-design implementation of the spectral clustering method.

Chapter 6 shows the synthesis results and metrics of the final design including hardware performance, resource utilization, and classification performance.

Chapter 7 concludes the work with discussion and some remarks.

Appendix consists of extra background information as well as guidlines for HW/SW co-design setup and use.

Chapter 2

Background

2.1 Hyperspectral Data Representation

Hyperspectral sensors collect multivariate discrete images in a series of narrow and contiguous wavelength bands. The resulting datasets contain numerous image bands, each of them depicting the scene as viewed with a given wavelength λ. This whole set of images can be seen as a three dimensional data cube where each pixel is characterized by a discrete spectrum related to the light absorption and/or scattering properties of the spatial region that it represents. Figure 2.1 shows an illustration of a hyperspectral image.

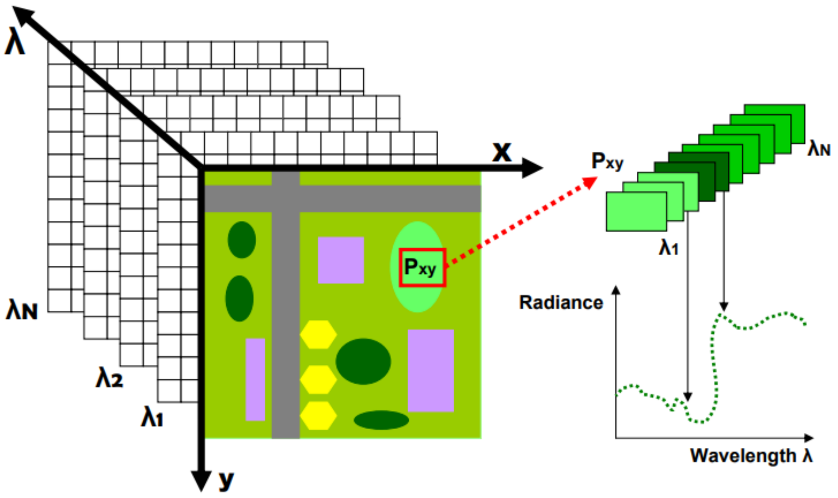


Figure 2.1: Illustration of a hyperspectral image [9].

The entire data I_λ can be seen as a three dimensional data cube formed by a set of N_z discrete 2D images $I_\lambda = I_{\lambda_1}, j = 1, \dots, N_z$. Each I_{λ_j} is formed by a set of N_p pixels where each pixel p represents the spatial coordinates in the image. Consequently, given a specific wavelength λ_j, $I_{\lambda_j}(p)$ is the radiance value of the pixel p on the waveband I_{λ_j}. As depicted in Figure 2.1, the spectral signature denoted by $I_\lambda(p)$ is the vector pixel p containing all the radiance values along the range of N_z wavelengths.

This signature provides insightful characteristics of the material represented by the pixel as shown for some materials in Figure 2.2.

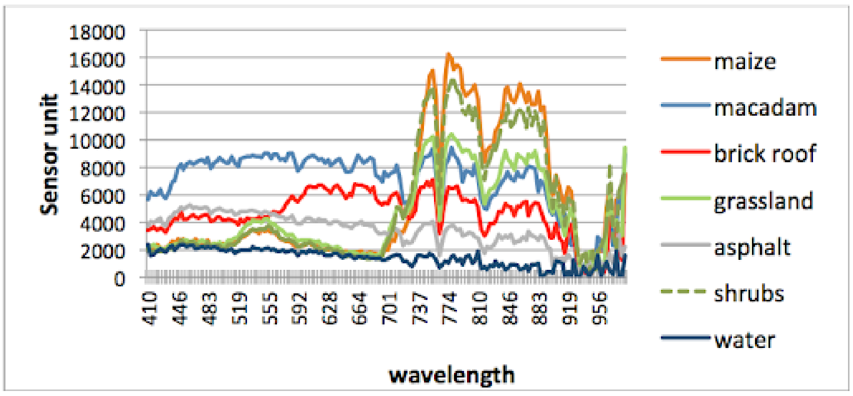


Figure 2.2: Spectral signatures of different materials [9].

The spectral space is important because it contains much more information about the surface of target materials than what can be perceived by the human eye. The spatial space is also important because it describes the spatial variations and correlation in the image and this information is essential to interpret objects in natural scenes. Hyperspectral image classification methods highly desired goals include automatic feature extraction using either one or both data information (spectral and spatial), in order to distinguish different materials for a meaningful application.

In this work, the HSI image for the input logic is organized such that all spectral components of one pixel are written in subsequent locations, followed by another pixel in a frame. Next row is formed for the next frame captured by the imager. Hence, bands 1 to 3 are written for pixel 1 which is part of frame 1, followed by components of pixel 2, and so on. This organization scheme is called band interleaved by pixel (BIP) format and is used throughout this work. An illustration is presented in Figure 2.3.

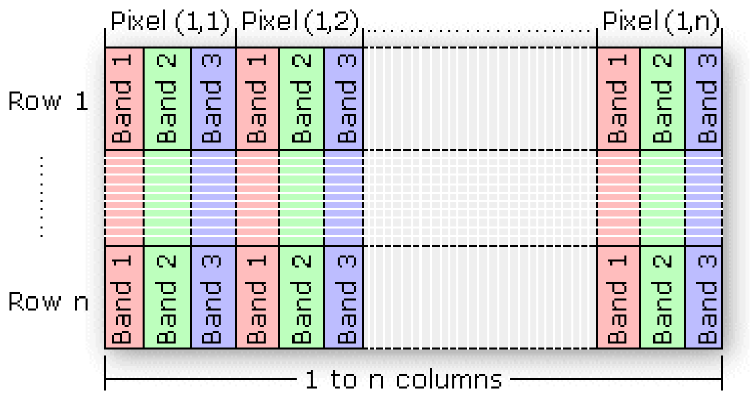


Figure 2.3: Hyperspectral Image BIP Storage Format.

2.2 HSI Classifcation Algorithms

As introduced in section 1.2, HSI classification algorithms can be categorized into supervised methods and unsupervised methods. Due to the noises and redundancy among spectral bands, many feature extraction, band selection, and dimension reduction techniques have been used for HSI classification in the past years. Supervised methods are popularly known to go through these techniques and follow it by training and labeling. The first common step among supervised methods consists of transforming the image to a feature image to reduce the data dimensionality and improve the data interpretability. This processing phase is optional and comprises techniques such as principal component analysis. In the training phase, a set of training samples in the image is selected to characterize each class. Training samples train the classifier to identify the classes and are used to determine the criteria for the assignment of a class label to each pixel in the image.

2.2.1 Convolution Neural Networks

Convolution Neural Networks (CNNs) are the most popular supervised learning classifiers. They are part of deep learning neural network architectures. Deep learning is part of the machine learning field which utilizes learning algorithms that derive meaning out of data by using a hierarchy of multiple layers. These multiple layers are constructed of neurons that mimic the neural networks of our brain.

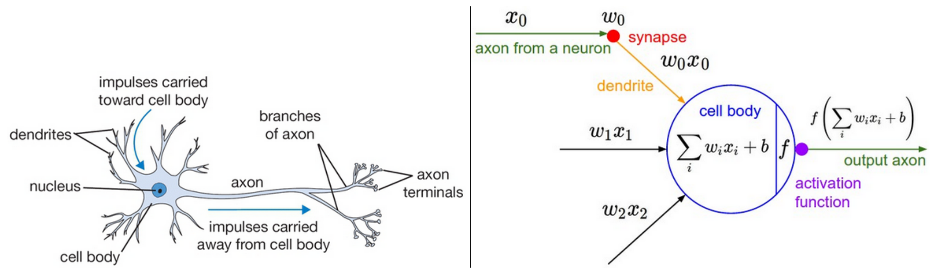


Figure 2.4: A cartoon drawing of a biological neuron (left) and its mathematical model (right) [10].

A single neuron is shown in Figure 2.4 (right) with an activation function. An activation function f takes a single number and performs a certain fixed mathematical operation on it. It defines the output of that node given an input or set of inputs. A relative example would be a standard computer chip circuit. It can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input. An example *neural network* would consist of a multiple of neurons where each neuron computes the scores s for different visual categories given the image using the formula $s = \sum_i w_i x_i$, where x is an input array of image pixels and w_i are parameters to be learned throughout the process via backpropagation.
The w_i parameters, also called weight vectors, are essentially the hidden layers of a neural network. Number of hidden layers determine the complexity of the network, which is high for high dimensional data like hyperspectral images.

2.2.2 Clustering Classification Algorithms

Clustering algorithms are a common technique used for unsupervised learning. Clustering refers to grouping objects into a set number of clusters whilst ensuring that the objects within each cluster group are similar to one another. In other words, we try to find homogeneous subgroups within the data such that data points in each cluster are as similar as possible according to a similarity measure, such as Euclidean-based distance. This technique is considered as an unsupervised learning method since it does not require to make use of the ground truth data information for the method's development. In addition, ground truth data may not be used to evaluate a clustering method's performance. In this work, two clustering algorithms are explored which make use of the popular k-means method.

One of the most used clustering algorithms is k-means due to its simplicity. This clustering method aims to partition the D-dimensional dataset $X = x_j, j = 1, \dots, D$ into clusters $C_i, i = 1, \dots, k$ such that each observation or data sample belongs to the cluster with the nearest mean. This results in a partitioning of the data space into disjoint sets $C_1 \dots C_k$ such that $D = C_1 \cup C_2 \cup \dots \cup C_k$.

Given a partition of the data into k clusters, the center μ_i (i.e, mean, center of mass) computed of each cluster i:

$$\mu_i = \frac{1}{n_i} \sum x_j \in C_i$$

For a well formed cluster, its points are close to its center according to a distance measure. We measure this with sum of squared error (SSE)

$$J = \sum_{i=1}^{K} \sum_{x_j \in C_i} ||x_j - \mu_i||^2$$

Hence, the main goal is to find the optimal partitioning C^* which minimizes the objective function J:

$$C^* = \underset{C^*=\{C_1,\dots,C_k\}}{argmin} \sum_{i=1}^{K} \sum_{x_j \in C_i} ||x_j - \mu_i||^2$$

Algorithm 1 Standard Kmeans Algorithm

```
Select $k$ random samples from $D$ as centers
Do
    for each example $x_i$,
        assign $x_i$ to cluster $C_j$ such that the distance $d(\mu_i, x_i)$ is minimized
    for each cluster j, update its cluster center such that
        $\mu_i = \frac{1}{n_i} \sum x_j \in C_i$
Until convergence or number of iterations is reached
```

Figure 2.5 shows an illustrative 2D example of the k-means algorithm application on a random plotted data points. In the example, for each iteration, we assign each train-

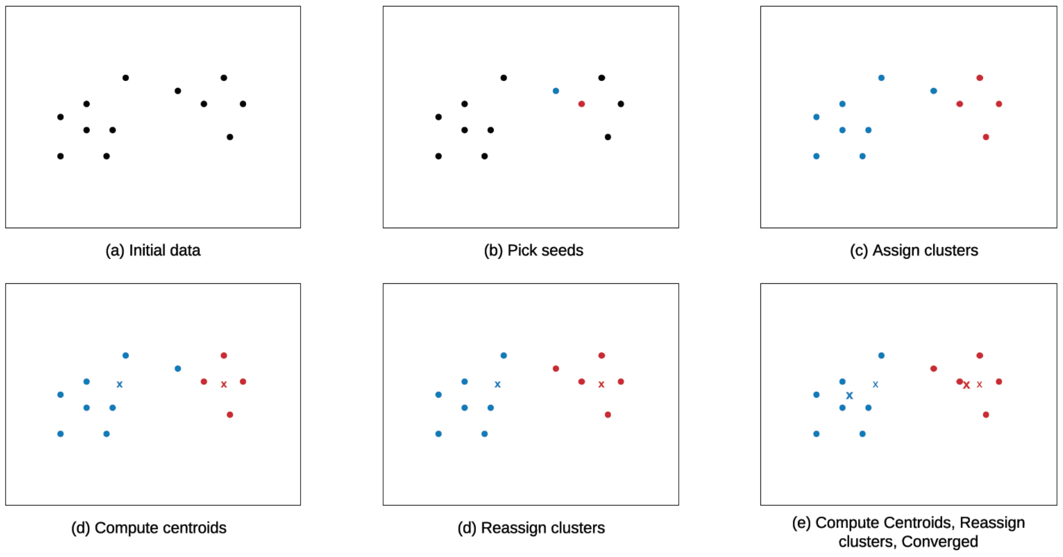


Figure 2.5: K-means algorithm 2D example.

ing example to the closest cluster centroid (shown by "painting" the training examples the same color as the cluster centroid to which is assigned); then we move each cluster centroid to the mean of the points assigned to it. It can be observed that the algorithm has converged when no further changes happen to the dataset.

2.3 State-of-the-art HSI Classification Algorithms

A review over the state-of-the-art hyperspectral image classification algorithms has been discussed in the specialization project, including implementations on FPGA [6]. However, due to further development of the two proposed methods in this thesis, their clustering method type is much specified and hence, the compared state-of-the-art methods on software development is redefined and restated.

In the literature, many HSI classification algorithms have been proposed and have achieved excellent performances on state-of-the-art public datasets described in section 4.2. However, most existing methods face challenges in dealing with large-scale hyperspectral image datasets due to their high computational complexity. Challenges for hyperspectral image classification also include:

- Curse of dimensionality resulting from large number of spectral dimensions.
- Scarcity of labelled training examples.
- Large spatial variability of spectral signature.

As discussed in section 1.2, HSI classification algorithms can be divided into supervised learning and unsupervised learning methods. A detailed discussion of different methods can be found in the specialization project. A quick summary is presented below along with the discussion of a new unsupervised state-of-the-art method.

2.3.1 Supervised vs. Unsupervised Learning Methods

The first step in a supervised learning method consists of transforming the image to a feature image to reduce the data dimensionality and improve the data interpretability. This processing phase comprises techniques such as principal component analysis (PCA). In the training phase, a set of training samples of the original image is selected and patched to characterize each class. The classifier is trained using the training samples with certain parameters that affect the rate of learning. During the training process, a patch of test samples (ground truth data) is used to validate the classifier learning progress and to find out if the classifier requires more training. Eventually, the classifier ends up with an assignment criteria for each class label of a certain hyperspectral image.

High accurate supervised methods incorporate Convolution Neural Networks (CNNs), discussed in 2.2.1. These networks act like convolution filters to extract features when applied to data. They can be designed in 1D, 2D or 3D forms. The difference is the structure of the input data and how the filter, also known as a convolution kernel or feature detector, moves across the data.
In the HSI context, it is evident from the literature that using just either 2D-CNN or 3D-CNN had a few shortcomings such as missing channel relationship information or very complex model, respectively [11]. The main reason is due to the fact that hyperspectral images are volumetric data and have a spectral dimension as well. The 2D-CNN alone isn't able to extract good discriminating feature maps from the spectral dimensions. Similarly, a deep 3D-CNN is more computationally complex and alone seems to perform worse for classes having similar textures over many spectral bands.
In Feb 2019, a hybrid-CNN model which overcomes these shortcomings of the previous models have been proposed in [11]. It consists of 3D-CNN and 2D-CNN layers which are assembled in such a way that they utilise both the spectral as well as spatial feature maps to their full extent to achieve maximum possible accuracy.

HSI classification based on such supervised methods provide excellent performance on standard datasets, e.g., more than 95% of the overall accuracy [11]. However, these existing methods face challenges in dealing with large-scale hyperspectral image datasets due to their high computational complexity [5]. Challenges for hyperspectral image classification also include the "curse of dimensionality", resulting from large number of spectral dimensions, and scarcity of labelled training examples as discussed earlier [12].

On the contrary, clustering-based techniques do not require prior knowledge and are commonly used for hyperspectral image unsupervised classification but still face challenges due to ***high spectral resolution*** and presence of ***complex spatial structures***. Apart from removal of noisy or redundant bands, the optimal selection of spectral band(s) is one of the major tasks during classification for HSI. To address the high spectral resolution problem, the most common clustering approach utilizes feature extraction methods for pre-processing such that one common subspace is produced for the whole data set where clustering takes place. In [13], k-means clustering method was used to form different clusters from the data obtained after applying principal component analysis (PCA) as a pre-processing technique. A local band selection approach is proposed in [14] where relevant set of bands is obtained using both relevancy and redundancy among the spectral bands. This approach takes care of redundancy among the bands by making use of interband distances.

2.3.2 Segment-based Clustering

One of the ways to tackle the complex spatial structures present in hyperspectral images is to make use of image segmentation to incorporate spatial information [15]. Image segmentation is a process in which an image is partitioned into multiple regions where pixels within a region share same characteristics and therefore have the same label [16]. These regions form a segmentation map that can be used as spatial structures for a spectral-spatial classification technique as described in [15]. Recently, different proposed frameworks made use of segment-based clustering classification for hyperspectral images [14, 17, 18]. The general idea is to apply clustering on the refined segments of a segmentation map instead of on pixels of hyperspectral images. In [19], spatial information is derived from the segmentation map, which is obtained by applying partitional clustering algorithms including projected clustering and correlation clustering. Projected clustering algorithms focus on identification of subspace clusters where each pixel is assigned to exactly one subspace cluster [20]. A similar approach is also proposed in [14] where k-means clustering is applied on the hyperspectral image to obtain an initial segmented map. The segmented map is further refined using multi-resolution segmentation for region merging. Furthermore, similar regions are merged by making use of their shape and spectral properties and hence refining the segmented map. Along the region merging stage and projected clustering stage, a local band selection approach. A more recent approach incorporating local band selection is proposed in [18] and its flowchart is found in Figure 2.6.

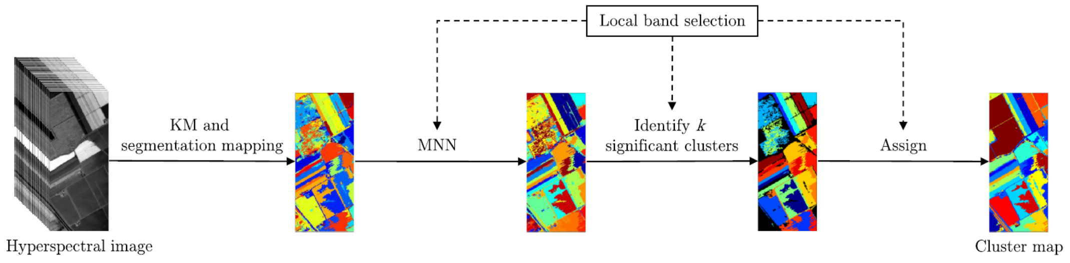

Local band selection
KM and segmentation mapping
MNN
Identify k significant clusters
Assign
Hyperspectral image
Cluster map

Figure 2.6: Flowchart of PCMNN. Dashed lines indicates that the local band selection approach is incorporated within these operations [18].

The obtained segmented map from the first stage is converted to a cluster map. The clusters are then merged by using the mutual nearest neighbour (MNN) information. In the last stage, the method identifies the k significant clusters using a criterion based on entropy. The final cluster map is obtained by assigning all the remaining clusters to these k significant clusters. The framework is considered the state-of-the-art for segment-based clustering methods for hyperspectral images, and is termed as, PCMNN (Projected Clustering using Mutual Nearest Neighbour) scoring highest accuracies on four HSI datasets. What is common among these techniques is that they use a three-step scheme in order (i) pre-segmentation of HSI, (ii) segmentation/region merging, and finally (iii) projected clustering for the HSI classification problem.

2.3.3 Method Choice

For HSI classification algorithms, in the context of the HYPSO mission objectives, the leading state-of-the-art supervised algorithm is the HybridSN CNN [11] with the following classification accuracies: 99.99% on Pavia University dataset, 99.81% on Indian Pines, and 100% on Salinas Scene. HybridSN seems to be the best choice for the HSI classification problem as it is the leading state-of-the-art algorithm, however, using clustering-based method implementation on FPGAs is preferred for a number of reasons:

1. It is generally difficult to generalize the architecture of a very complex deep learning model. Different layers have different parameters. The main challenge is that CNN architectures do not usually have identical layer parameters, which increases the difficulty of designing generic hardware modules for the FPGA. For example, there are 1-D convolutional layers with different kernel sizes such as 3x1 and 5x1.
2. We notice that the recent developed CNNs are biased towards the 3 datasets (Pavia University, Indian Pines, and Salinas Scene), which are land-cover based datasets, and hence might not perform well for other datasets such as Samson, Urban, and Jasper Ridge which incorporate "water" as one of the classes. CNNs might also not perform well on such datasets because images may not be of the expected high quality or the dataset itself is small.
3. The highest classification reached for the HSI classification problem is approximately 100% on the 3 datasets. Hence, there is a tendency in research for improving the other classification methods, e.g. clustering, so that it performs better for other datasets, including small datasets or unlabeled datasets which is not efficient for CNNs to learn from. In this study, 8 HSI datasets are experimented on, which are both land-cover and water based images.
4. The simple control flow and inherent parallelism of distance computations makes clustering suitable for dedicated hardware implementations such as FPGAs and this found to be proven in [21].

Based on the specialization project [6] and the method choice above, this thesis presents implementations of clustering-type unsupervised classification algorithms for hyperspectral images. Two algorithms are implemented in software; a novel segment-based clustering algorithm and a spectral clustering algorithm. Furthermore, based on coming experiments and analysis in Chapter 4, the spectral clustering method is implemented for FPGA. Necessary background information for both hardware and software implementations is detailed in the sections below.

2.4 Binary Partition Trees and HSI

The implemented segment-based clustering method makes use of binary partition trees (BPTs) for segmentation. BPT is a hierarchical region-based representation of relationships among data. It is a set of hierarchical regions of data stored in a tree structure [22]. The tree structure consists of tree nodes representing image regions, and the branches represent the inclusion relationship among the nodes. Figure 2.7 is an illustration of a hierarchical representation of regions using BPT. In this tree representation, three types of nodes can be found: Firstly, leaves nodes representing the original regions of the initial image partition; secondly, the root node representing the entire image support and finally, the remaining tree nodes representing regions formed by the merging of their two child nodes corresponding to two adjacent regions. Each of these non leaf nodes has at most two child nodes, this is why the BPT is defined as binary.

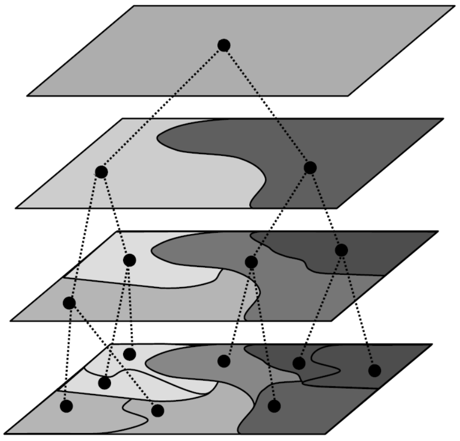


Figure 2.7: Example of hierarchical region-based representation using BPT [9].

In the context of hyperspectral images, a BPT representation is generated using (i) a given region model and (ii) a region merging criterion, both recently developed by [9]. Given the initial partition of small regions, the BPT is constructed in such a way that the most meaningful regions of the images are represented by nodes. The leaf nodes correspond to the initial partition of the image. From this initial partition, an iterative bottom-up region merging algorithm is applied by keeping track of the merging steps where the most similar adjacent regions are merged at each iteration until only one region remains. This last region represents the whole image and is the root of the tree. The creation of BPT relies on three important notions:

1. The method obtaining the initial partition, also called, pre-segmentation.
2. The region model M_{R_i} which defines how regions are represented and how to model the union of two regions.
3. The merging criterion $O(M_{R_i}, M_{R_j})$, which defines the similarity of neighboring regions as a distance measure between the region models M_{R_i} and M_{R_j} and involves determining the order in which regions are going to be merged.

In [9], different region models with their compatible region merging criterion were described to construct the BPT.

2.4.1 Pre-segmentation using Watershed Method

A watershed is a transformation defined on a grayscale image. To perform watershed segmentation, a 2D image is viewed as a topographic map that shows hills and valleys as it can be observed in Figure 2.8. In that map/image, the value of each pixel corresponds to the elevation at that pixel point. Thus, dark areas (low pixel value) are valleys, while bright areas (high pixel value) are hills. The watershed transform finds catchment basins (or dark areas) in this landscape. For example, when it rains on a mountain, water slides onto two different paths (two catchment basins). Thus, the ridgeline at the top of the mountain is also a line that divides two catchment basins so that each basin is associated with one minimum and hence, divides the image as in Figure 2.8 [23].

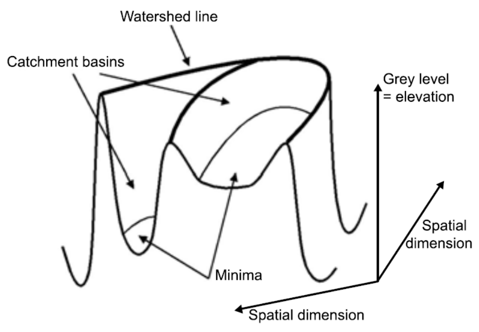


Figure 2.8: Topographic representation of a 2D mountain image [23].

The watershed transformation is usually applied to the gradient function of the image. An image gradient is a directional change in the intensity or color in an image, it defines transitions between regions such that the borders between reigons have high transitional values [23]. A gradient on a multivariate function can be obtained in different ways. One way is to calculate on each image channel a modulus of a gradient, and to take the sum or the supremum of the gradients [24]. Another way is to use vectorial gradients based on distance between vector pixels, distances can be Euclidean-based [25]. The gradient function used in this work is based on obtaining the supremum of the gradients. To obtain an oversegmentation, the gradient is obtained for each spectral band and then the spatial gradient is computed as the maximum gradient value of all the bands. [23] provides thorough explanation for watershed segmentation on hyperspectral images. The output of the watershed transform is a partition of the image composed of regions (sets of pixels connected to the same local minimum) and of watershed pixels (WHEDs, the borders between the regions). By the end of this stage, an initial segmented map of oversegmented regions is obtained as shown in Figure 3.1.

2.4.2 BPT Building

The BPT leaf nodes correspond to the initial partition of the image. From this initial partition, an iterative bottom-up region merging algorithm is applied by keeping track of the merging steps where the most similar adjacent regions are merged at each iteration until only one region remains. Figure 2.9 depicts the BPT building process.

The framework makes use of the first-order parametric model for region modelling due to its simplicity in definition leading to simple merging order process [9].
Given a hyperspectral region R formed by N_{R_p} spectra containing N_n different radiance values, the first-order parametric model M_R is defined as a vector of N_n components which corresponds to the average of the values of all spectra $p \in R$ in each band λ_i shown in Figure 2.10.

$$M_R(\lambda_i) = \frac{1}{N_{R_p}} \sum_{j=1}^{N_{R_p}} H_{\lambda_i}(p_j) \qquad i \in [1, \dots, N_n] \tag{2.1}$$

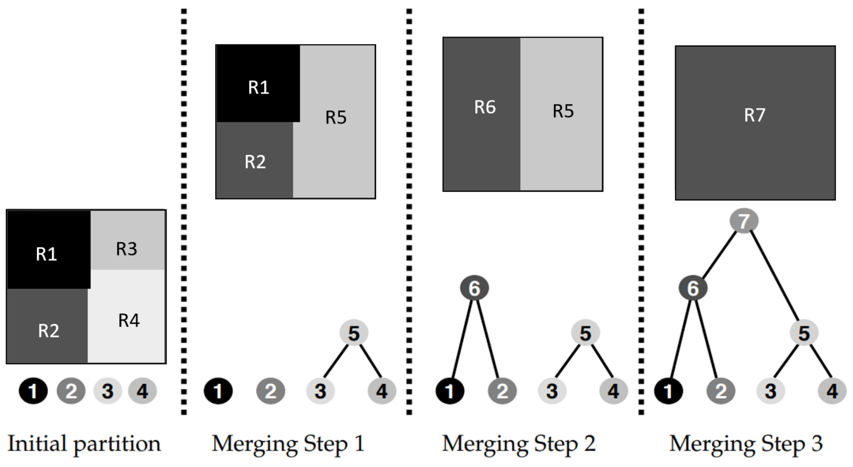


Figure 2.9: BPT construction using a region merging algorithm [9].

Note that $H_{\lambda_i}(p_j)$ represents the radiance value in the wavelength λ_i of the pixel whose spatial coordinates are p_j.

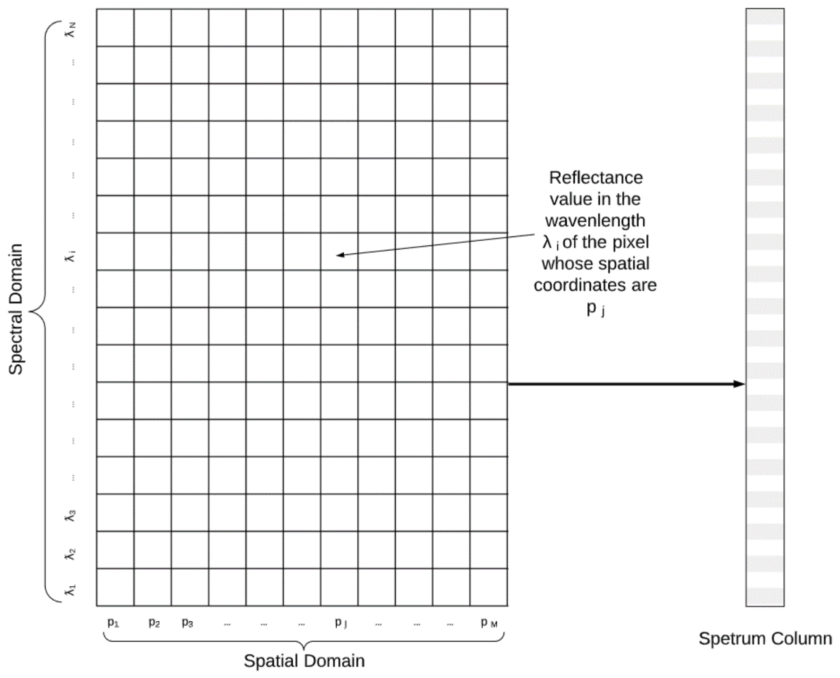

Spectral Domain
λ N
λ i
λ 3
λ 2
λ 1
p1
p2
p3
p j
p M
Spatial Domain
Reflectance value in the wavenlength λ i of the pixel whose spatial coordinates are p j
Spetrum Column

Figure 2.10: A grid representing the set of spectra for a region R containing M pixels and modelled into one column of spectra.

Using the first-order parametric model (2.1), a merging criterion is defined as the spectral angle distance, dSAD, between the average values of any two adjacent regions:

$$O(M_{R_a}, M_{R_b}) = d_{SAD} = arccos\left(\frac{R_a^T R_b}{||R_a||||R_b||}\right), \tag{2.2}$$

where R_a, R_b are two different regions and M_{R_a}, M_{R_b} are their corresponding spectrum column region model.
The region merging stage runs until there are no more mutual best neighbours available for merging. Further, before merging regions, small and meaningless regions in the initial partition of the previous stage may result in a spatially unbalanced tree during BPT construction. Those regions are prioritized to be merged first in the region merging order by determining whether they are smaller than a given percentage (typically 15%) regions created by the merging process of the average region size of the initial partition [9].

2.4.3 BPT Pruning

Until this point, we have obtained a BPT representation of the hyperspectral image incorporating its spatial and spectral features. The next step is to process the BPT such that we get a partition featuring the N most dissimilar regions created during the construction of the binary partition tree. This can be done by extracting a given number of regions P_R (pruned regions), hence pruning the tree. Different BPT pruning strategies lead to different results [26]. Furthermore, there exists several BPT pruning strategies suited for a number of applications like classification, segmentation, and object detection [27, 28].

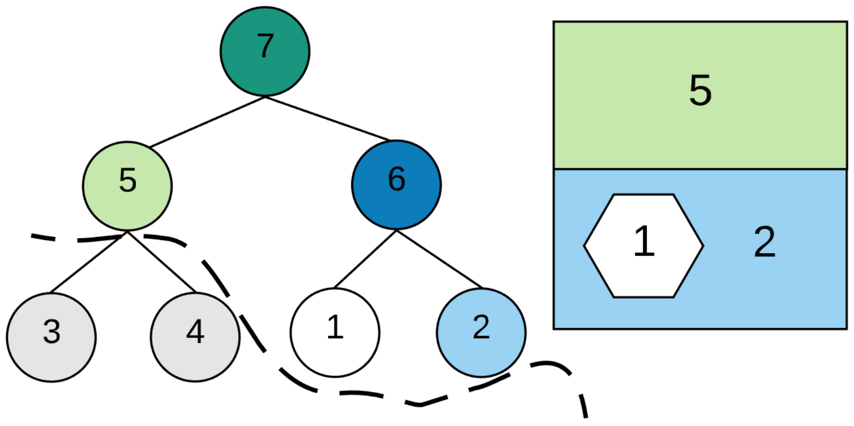


Figure 2.11: Region-based pruning of the Binary Partition Tree using $P_R = 3$

For the purpose of this stage of the proposed framework, we further prune the BPT for the segmentation goals by using a simple pruning strategy based on the number of regions [26]. The region-based pruning provides the last P_R regions remaining before the completion of the BPT building process. This can be obtained by traversing the tree in an inverse order to its construction and stop when the desired number of regions $P_R \geq 0$ has been reached. In other words, the final segmentation map will be composed of the P_R merged regions according to the merging order during BPT construction. For instance, if the building of the nodes was done in the order R4 $\Rightarrow$ R5 $\Rightarrow$ R6 $\Rightarrow$ R7 and $P_R = 3$, the BPT will be pruned after the merging of regions R3 and R4 into R5 as it can be seen in Figure 2.11.

The main advantage of this pruning strategy is its simplicity and it also shows whether the BPT was constructed in a meaningful way or not since the last P_R regions are the most dissimilar regions. By the end of this stage, a well refined segmentation map of the hyperspectral image is obtained which can be further used by a k-means filtering algorithm.

2.5 Filtering Algorithm

K-means algorithm complexity goes up linearly in k centres, the number of data points and the number of dimensions in a dataset, and the number of iterations [29]. This is because a straightforward implementation of k-means standard algorithm can be quite slow due to the cost of computing nearest neighbors; we want to find the minimum argument for each cluster group and calculate the distances of each point in a cluster to the centroid of that cluster. Thus, it becomes infeasible in high dimensional spaces with many data points such as hyperspectral images.

Furthermore, getting the argmin of the objective function J defined in section 2.2.2 is computationally expensive for hyperspectral images [29]. In [30], a filtering algorithm was developed for a multidimensional binary search tree, called a *k-d tree*. A *k-d tree* is a space partitioning tree data structure for organizing data points in a K-Dimensional space. The main goal of the filtering algorithm is to prune (filter) down the search space for the optimal partition such that the computation burden is reduced [21]. In this work, we make use of the filtering algorithm for another binary tree structure that allows flexibility in region splitting (BPT discussed in section 2.4).

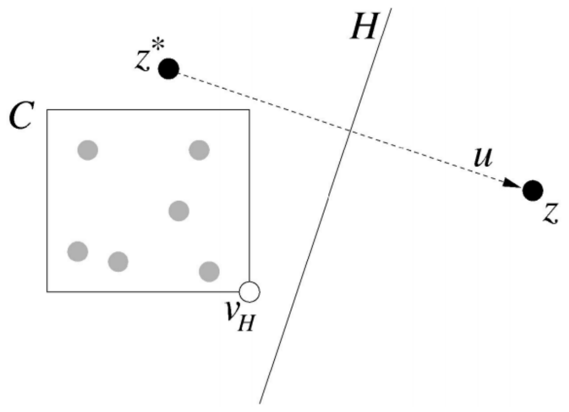


Figure 2.12: Candidate z is pruned because C lies entirely on one side of the bisecting hyperplane H [8].

Assume that the hyperspectral image is segmented into regions using a segmentation method and is represented by a binary tree structure such that the leaves of the tree represent the partitions within the HSI cube. At the initial iteration of clustering, the regions of the segmentation map act as a base for the algorithm to start from. During clustering, these regions are either differentiated or joined by assigning them cluster labels as the tree is traversed for a number of iterations.
Algorithm 2 shows pseudo code for one iteration of the filtering algorithm. Each tree

node u represents a region and is associated with the bounding box (C) information as well as the number of pixels ($count$) and the vector sum of the associated pixels (the weighted centroid, $wgtCent$) which is used to update the cluster centres when each iteration of the algorithm completes. The actual centroid is just $wgtCent/count$. In addition, for each node of the tree, we maintain a set of possible candidate centers Z that is propagated down the tree. This is defined to be a set of center points that can be the nearest neighbor for some pixel within a tree node. It follows that the candidate centers for the root node consist of all k centers.

For a tree node u, the closest candidate center $z^* \in Z$ to the midpoint of the bounding box is found during clustering. The closest centre search involves the computation of the Euclidean distances. The remaining $z \in Z \setminus \{z^*\}$ is pruned to the following node down the tree if no part of C is closer to z than the closest centre z^* since it is not the nearest center to the pixels within C. Figure 2.12 shows an illustration where the set of points are enclosed by the closed cell C and z^* is the closest center point to the points in C. Hence, the center z is not further considered as a center since no part of C is closer to z than to z^* and z is removed from the center candidate list for the closed cell C. If u is an internal node, we recurse on its children. The centre update occurs if a leaf is reached or only one candidate remains.

Algorithm 2 The filtering algorithm introduced in [30]

```
function Filter (kdNode $u$, CandidateSet $Z$) {
$C \leftarrow u.C$
**if** ($u$ is a leaf) {
        $z^* \leftarrow$ the closest point in Z to $u.point$
        $z^*.wgtCent \leftarrow z^*.wgtCent + u.point$
        $z*.count \leftarrow z^*.count + 1$ }
**else** {
        $z^* \leftarrow$ the closest point in Z to C's midpoint
        **for all** ($z \in Z \setminus \{z^*\}$) **do**
            **if** z.isFarther($z^*, C$) **then** $Z \leftarrow Z \setminus \{z\}$
        **end for**
        **if** ($|Z| = 1$) {
            $z^*.wgtCent \leftarrow z^*.wgtCent + u.wgtCent$
            $z*.count \leftarrow z^*.count + u.count$ }
        **else** {
            Filter($u.left$, Z)
            Filter($u.right$, Z) }
    }
}
for all ($z \in Z$)
        $z \leftarrow z.wgtCent/z.count$
```

The $isFarther$ function of Algorithm 2 checks whether any point of a bounding box C is closer to z than to z^*. The function returns true if the sum of squares difference between the candidate z and the closest candidate z^* is greater than 2 × the sum of squares difference between either boundary points (high or low) and the closest candidate center z^*. On termination of the filtering algorithm, center z is moved to the centroid of its associated points, that is, $z \leftarrow z.wgtCent/z.count$.

2.6 Spectral Clustering

Spectral clustering techniques are widely used, due to their simplicity and empirical performance advantages compared to other clustering methods, such as k-means [31]. This type of clustering can be solved efficiently by standard linear algebra methods. The basic idea is to represent each data point as a graph-node and thus transform the clustering problem into a graph-partitioning problem. Thus, the goal is to identify communities of nodes in a graph based on the edges connecting them. A typical implementation consists of two fundamental steps: 1. Building a similarity graph and 2. Finding an optimal partition of the constructed graph.

2.6.1 Building the Similarity Graph

Given a set of data point samples $x_1, \ldots, x_n$, the method first builds a similarity graph $G = \{Vertices, Edges\}$ such that each vertex represents a sample x_i and the edge to another point x_j is weighted by the similarity w_{ij} where w_{ij} is positive or larger than a certain threshold. The problem of clustering can now be reformulated using the similarity graph: we want to find a partition of the graph such that the edges between different groups have very low weights where points in different clusters are dissimilar from each other, and the edges within a group have high weights such that points within the same cluster are similar to each other.
In general, the corresponding similarity (or affinity) matrix W can be denoted as

$$W_{ij} = \exp \frac{-||x_i - x_j||_2^2}{2\sigma^2}, \qquad i, j = 1, 2, \ldots, n, \tag{2.3}$$

where σ is the width of the neighbors of the samples and n is the number of graph nodes.

2.6.2 Finding an Optimal Partition

Let A and B represent a bipartition of Vertices, where $A \cup B =$ Vertices and $A \cap B = \emptyset$. The simplest and most direct way to construct a partition of the graph is to solve the minimum cut (mincut) problem. For a given number k of subsets, the mincut approach simply consists in choosing a partition $A_1, \ldots, A_k$ which minimizes cut(A,B) = $\sum_{i \in A, j \in B} W_{ij}$. Hence, cut(A,B) denotes the sum of the weights between A and B. In this case, there are two partitions, $k = 2$, and the mincut problem is relatively easy to solve [31]. However, in practice the solution often does not lead to satisfactory partitions. The problem is that in many cases, the solution of mincut simply separates one individual vertex from the rest of the graph. Figure 2.13 illustrates such case. Assuming the edge weights are inversely proportional to the distance between the two nodes, it can be observed that the cut partitioning node n1 or n2 will have a very small value. In fact, any cut that partitions out individual nodes on the right half will have smaller cut value than the cut that partitions the nodes into the left and right halves. Of course this is not what we want to achieve in clustering, as clusters are expected to represent reasonably large groups of points.

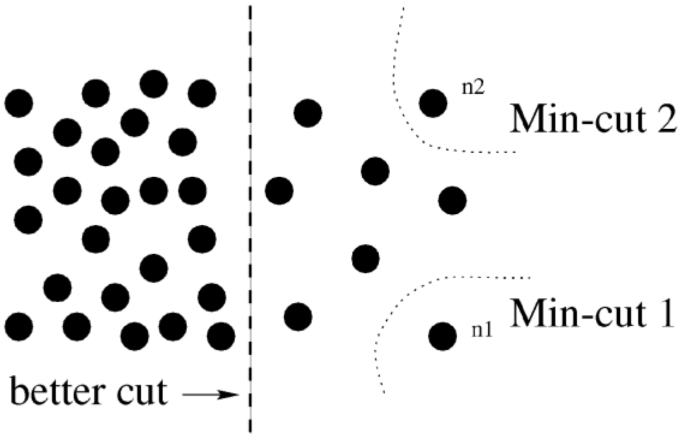


Figure 2.13: A case where minimum cut gives a bad partition [32].

One way to circumvent this problem is to explicitly request that the partitions $A_1, \ldots, A_k$ are "reasonably large". This is done by transforming the problem into an objective function, called normalized minimum cuts (NCuts), such that the size of a partition A of a graph is measured by the weights of its edges, its volume $vol(A)$. The volume of a partition is defined as the sum of the degrees within that set: $vol(A) = \sum_{i \in A} d_{ii}$ where $d_{ii} = \sum_j W_{ij}$. Furthermore, D is an $N \times N$ diagonal matrix with d_{ii} on its diagonal. Hence, the normalized cut between A and B can be considered as follows:

$$NCut(A,B) = \frac{cut(A,B)}{vol(A)} + \frac{cut(B,A)}{vol(B)} \tag{2.4}$$

The following mathematical background analysis and steps are based on [32] with some additional derivations done in this thesis.
Let x be an indicator vector of the size N, $x_i = 1$ if node i is in A and -1, otherwise. Using the definitions of x and d, we can rewrite $NCut(A,B)$ as:

$$NCut(A,B) = \frac{\sum_{(x_i>0,x_j<0)} -w_{ij}x_ix_j}{\sum_{x_i>0} d_i} + \frac{\sum_{(x_i<0,x_j>0)} -w_{ij}x_ix_j}{\sum_{x_i<0} d_i}.$$

Let m be a constant; $m = \frac{\sum_{x_i>0} d_i}{\sum_i d_i}$, and $\mathbf{1}$ be an $N \times 1$ vector of all ones. Using the fact that $\frac{1+x}{2}$ and $\frac{1-x}{2}$ are indicator vectors for $x_i > 0$ and $x_i < 0$, respectively, $4 \times [NCuts]$ can be written as:

$$\begin{aligned} &= \frac{(1+x)^T(D-W)(1+x)}{m1^TD1} + \frac{(1-x)^T(D-W)(1-x)}{(1-m)1^TD1} \\ &= \frac{(x^T(D-W)x + 1^T(D-W)1)}{m(1-m)1^TD1} + \frac{2(1-2k)1^T(D-W)x}{m(1-m)1^TD1} \end{aligned} \tag{2.5}$$

This can be shown by the following: Let us consider there are 3 nodes such that 2 nodes in partition A and 1 node in partition B. Then, $x = \{1, 1, -1\}$ where $x_i = 1$ indicates that the node belongs to A, otherwise, it belongs to B. Then the first numerator part of NCut(A,B) is:

$$\sum_{(x_i>0,x_j<0)} -w_{ij}x_ix_j = -w_{13}x_1x_3 - w_{23}x_2x_3 = w_{13} + w_{23}.$$

Hence, $4 \times [NCuts]$ (1st part numerator) can be written as:

$$(1+x)^T(D-W)(1+x) => \begin{bmatrix} 2 & 2 & 0 \end{bmatrix} \begin{bmatrix} w_{12}+w_{13} & -w_{12} & -w_{13} \\ -w_{21} & w_{21}+w_{23} & -w_{23} \\ -w_{31} & -w_{32} & w_{31}+w_{32} \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix}$$

This is equal to $4 \times (w_{13} + w_{23})$ as expected. A similar argument can be made for the 2nd part of the numerator of $4 \times [NCuts]$.
As for the denominator part,

$$m = \frac{\sum_{x_i>0} d_i}{\sum_i d_i} = \frac{w_{11}+w_{12}+w_{21}+w_{22}+w_{13}+w_{23}}{w_{11}+w_{12}+w_{13}+w_{21}+w_{22}+w_{13}+w_{23}+w_{33}},$$

then,

$$m1^TD1 = \frac{w_{11}+w_{12}+w_{21}+w_{22}+w_{13}+w_{23}}{w_{11}+w_{12}+w_{13}+w_{21}+w_{22}+w_{13}+w_{23}+w_{33}} \times$$
$$(w_{11}+w_{12}+w_{13}+w_{21}+w_{22}+w_{13}+w_{23}+w_{33}) = \sum_{x_i>0} d_i.$$

Let

$$\alpha(x) = x^T(D-W)x,$$
$$\beta(x) = 1^T(D-W)x,$$
$$\gamma = 1^T(D-W)1,$$

and

$$Q = 1^TD1,$$

then, Equation 2.5 can be further expanded as:

$$= \frac{(\alpha(x)+\gamma)+2(1-2m)\beta(x)}{m(1-m)Q}$$
$$= \frac{(\alpha(x)+\gamma)+2(1-2m)\beta(x)}{m(1-m)Q} - \frac{2(\alpha(x)+\gamma)}{Q} + \frac{2\alpha(x)}{Q} + \frac{2\gamma}{Q}.$$

Dropping the last constant term such that it equals to 0, we get

$$= \frac{(1-2m+2m^2)(\alpha(x)+\gamma)+2(1-2m)\beta(x)}{m(1-m)Q} + \frac{2(\alpha(x)+\gamma)}{Q}$$

$$= \frac{\frac{1-2m+2m^2}{(1-m)^2}(\alpha(x)+\gamma)+\frac{2(1-2m)}{(1-m)^2}\beta(x)}{\frac{m}{(1-m)}Q} + \frac{2\alpha(x)}{Q}.$$

Since $\gamma = 0$, then $\frac{2b\gamma}{bQ} = 0$ and letting $b = \frac{m}{(1-m)}$, it becomes

$$= \frac{(1+b^2)(\alpha(x)+\gamma)+2(1-b^2)\beta(x)}{bQ} + \frac{2b\alpha(x)}{bQ}$$

$$= \frac{(1+b^2)(\alpha(x)+\gamma)+2(1-b^2)\beta(x)}{bQ} + \frac{2b\alpha(x)}{bQ} - \frac{2b\gamma}{bQ}.$$

Substituting back, it becomes

$$= \frac{(1+b^2)(x^T(D-W)x+1^T(D-W)1)}{b1^TD1} + \frac{2(1-b^2)1^T(D-W)x}{b1^TD1} + \frac{2b(x^T(D-W)x)}{b1^TD1} - \frac{2b1^T(D-W)1}{b1^TD1}$$

$$= \frac{(1+x)^T(D-W)(1+x)}{b1^TD1} + \frac{b^2(1-x)^T(D-W)(1-x)}{b1^TD1} - \frac{2b(1-x)^T(D-W)(1+x)}{b1^TD1}$$

$$= \frac{(1+x)^T(D-W)(1+x)}{b1^TD1} + \frac{b^2(1-x)^T(D-W)(1-x)}{b1^TD1} - \frac{2b(1-x)^T(D-W)(1+x)}{b1^TD1}.$$

Using matrix transpose properties, we get

$$= \frac{[(1+x)-b(1-x)^T(D-W)[(1+x)-b(1-x)]}{b1^TD1}. \tag{2.6}$$

The goal now is to formulate expression 2.6 such that we find a minimum objective function with conditions.
Setting $y = (1+x)-b(1-x)$, where $(1+x)$ is a vector with all the nodes in the partition A $(x_i > 0)$ and the vector $(1-x)$ is otherwise $(x_i < 0)$. And since $(A-B)^T = A^T - B^T$, we can see that

$$y^TD1 = \sum_{x_i>0} d_i - b\sum_{x_i<0} d_i = \sum_{x_i>0} d_i - (\frac{m}{1-m})\sum_{x_i<0} d_i$$

$$= \sum_{x_i>0} d_i - \frac{\sum_{x_i>0} d_i}{\sum_{x_i<0} d_i}\sum_{x_i<0} d_i = 0.$$

We can deduce that

$$y^TDy = \sum_{x_i>0} d_i + b^2\sum_{x_i<0} d_i = b\sum_{x_i<0} d_i + b^2\sum_{x_i<0} d_i$$

$$= b(\sum_{x_i<0} d_i + b\sum_{x_i<0} d_i)$$

$$= b1^TD1$$

Hence, the minimum objective function for expression 2.6 becomes

$$min_x Ncut(x) = min_y \frac{y^T(D-W)y}{y^TDy} \tag{2.7}$$

with the condition $y^TD1 = 0$.
The right hand side of Equation 2.7 is the Rayleigh quotient if y is relaxed to take on

real values such that the matrices and vectors are real [32]. Hence, the problem can be minimized by solving the generalized eigenvalue system:

$$(D-W)y = \lambda D y. \tag{2.8}$$

Let $z = D^{\frac{1}{2}}y$, then the generalized eigenvalue system 2.8 becomes

$$(D-W)D^{-\frac{1}{2}}z = \lambda D D^{-\frac{1}{2}}z,$$

multiply $D^{-\frac{1}{2}}$ on both sides,

$$D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}}z = \lambda D^{-\frac{1}{2}} D D^{-\frac{1}{2}}z,$$

$$D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}}z = \lambda z. \tag{2.9}$$

We reach the standard eigensystem 2.9 which is to be minimized.
Note that $D-W$ is considered the "Laplacian Matrix" L. The Laplacian is just another matrix representation of a graph which has the following properties:

- L is a positive semidefinite matrix.
- The smallest eigenvalue of an eigensystem of L is 0, the corresponding eigenvector is the constant 1 vector.
- L has n non-negative, real-valued eigenvalues such that $0 = \lambda_1 \leq \lambda_2 \leq \ldots \lambda_n$.

According to spectral graph theory, an approximate resolution of Equation 2.7 can be considered as thresholding the eigenvector corresponding to the second smallest eigenvalues of the normalized Laplacian L. The second smallest nonzero eigenvalue is called the spectral gap. The spectral gap gives us some notion of the density of the graph. If this graph was densely connected (all pairs of the n nodes had an edge), then the spectral gap would be n, where n is number of nodes.

The direct relation between the eigenvalues of a Laplacian matrix and clustering is summarized below:

- When the nodes in a graph are completely disconnected, then all eigenvalues are 0.
- Otherwise, the 1st eigenvalue is always 0 since there is only one connected component which is the graph itself.
- The 2nd eigenvalue approximates the minimum graph cut needed to separate the graph into two connected components.
- Near zero 2nd eigenvalue shows that there is almost a separation of two components.
- Each value in the 2nd eigenvector gives us information about which side of the cut a node belongs to, either positive or negative.

Algorithm 3 summarizes shows the established spectral clustering algorithm.

Algorithm 3 Spectral Clustering

```
Input: dataset $S = \{s_1, s_2, \ldots, s_n\}$, number of clusters $k$
Output: $k$-clustering of $S$
$W_{ij} = \exp \frac{-||x_i - x_j||_2^2}{2\sigma^2}, \qquad i, j = 1, 2, \ldots, n$
$D_{ij} = d_{ii} = \sum_j W_{ij}$
$L = D^{-\frac{1}{2}}(D - W)D^{-\frac{1}{2}}$
Find $X = SmallestEigenVectors(L, k)$
$Y = Normalize(X)$
$k = Cluster(Y)$
```

In summary, as shown in Algorithm 3, once the affinity/similarity matrix is computed, the graph Laplacian matrix can be constructed. The first k eigenvectors of L are then normalized (to be considered as NCuts not mincut) and clustered using any k-clustering algorithm such as k-means discussed in Section 2.2.2.

2.6.3 Eigenvalue and Eigenvector Decomposition

This subsection focuses on the eigenanalysis of the Laplacian matrix L. Recall that eigenvalues are defined as roots of the characteristic equation $det(A - \lambda I_m) = 0$, where A is any $(m \times m)$ matrix. In matrix format, the eigenvalue equation for A is defined as

$$AU = U\Lambda,$$

where we put together the set of eigenvectors of A in a matrix denoted U. Each column of U is an eigenvector of A. The eigenvalues are stored in a diagonal matrix (denoted Λ), where the diagonal elements give the eigenvalues (and all the other values are zeros). Multiplying both sides by U^{-1}, A can be also written as

$$A = U\Lambda U^{-1}.$$

From section 2.6.2, recall that L is a positive semi-definite. This means that it can be obtained as the product of a matrix multiplied by its transpose [33]. This further implies that a positive semi-definite matrix is always symmetric since the product of a matrix and its transpose results in a symmetric matrix $(m \times n) \times (n \times m) = (m \times m)$. Thus, L can be obtained as:

$$L = X^T X$$

for a certain matrix X containing real numbers. The important properties of L is that its eigenvalues are always positive or null, and its eigenvectors are pairwise orthogonal when their eigenvalues are different. Hence, since eigenvectors corresponding to different eigenvalues are orthogonal, it is possible to store all the eigenvectors in an orthogonal matrix (recall that a matrix is orthogonal when the product of this matrix by its transpose is the identity matrix) [34]. In other words, $UU^T = I$ and this implies that $U^-1 = U^T$. Hence, L becomes

$$L = U\Lambda U^T,$$

which is called the diagonalization of L.
The goal is to obtain the diagonalization of L. This can be done by making use of the singular value decomposition (SVD).

Singular Value Decomposition

According to [35], for a rectangular matrix A $(m \times n)$, let $p = min\{m, n\}$ the SVD theorem states that

$$A = U \Sigma V^T, where$$

- $U = (u_1, \dots, u_m) \in \mathbb{R}^{m \times m}$ is orthogonal.
- $V = (v_1, \dots, v_n) \in \mathbb{R}^{n \times n}$ is orthogonal.
- $\Sigma = diag(\sigma_1, \dots, \sigma_p) \in \mathbb{R}^{m \times n}, \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0.$

σ_i are called the singular values, u_i are the left singular vectors, and v_i are the right singular vectors. The singular values, σ_i, are the square roots of the eigenvalues of $A^T A$ and of AA^T (these two matrices have the same eigenvalues).

Corollary 1 *If A is a symmetric matrix the singular values are the absolute values of the eigenvalues of $A : \sigma_i = |\lambda_i|$ and the columns of $U = V$ are the eigenvectors of A.*

Proof: If A is symmetric then $AA^T = A^T A = A^2$ and U, V, Σ are square matrices. The eigenvectors of A are also the eigenvectors of A^2 with squared corresponding eigenvalues. The singular values are the absolute values of the eigenvalues of A. $\square$

This implies that for the symmetric matrix L, $L = U \Lambda U^T$ is the SVD of L where U and Λ are the eigenvalues and the eigenvectors of L, respectively. Hence, finding the SVD of the Laplacian matrix corresponds to finding its eigenvalue decomposition.

Jacobi Method

Jacobi methods are quite popular in hardware implementation of SVD solution because the computation can be done in parallel. The main idea of a Jacobi method in SVD context is to get the eigenvectors of A using a sequence of "Jacobi rotations" to diagonalize it [36]. Diagonalizing a matrix essentially means that the off-diagonal entries of that matrix are zeros. An off-diagonal entry is any entry of a matrix that is not on its main diagonal. For example, we may define a diagonal matrix as being a square matrix whose off-diagonal entries are all equal to zero. A Jacobi rotation matrix, $J(p, q, \theta)$, is defined as:

$$\begin{array}{cc}
\left[\begin{array}{ccccccc}
1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & & \vdots & & \vdots \\
0 & \cdots & c & \cdots & s & \cdots & 0 \\
\vdots & & \vdots & \ddots & \vdots & & \vdots \\
0 & \cdots & -s & \cdots & c & \cdots & 0 \\
\vdots & & \vdots & & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & \cdots & 0 & \cdots & 1
\end{array}\right] &
\begin{array}{c} \\ \\ p \\ \\ q \\ \\ \\ \end{array} \\
\begin{array}{ccccccc} & & p & & q & & \end{array} &
\end{array}$$

where $c = cos\theta$ and $s = sin\theta$. The matrix J is accordingly an identical matrix, except its four elements c, s, c, and $-s$. This matrix, when applied as a similarity transformation to a symmetric matrix A $(n \times n)$, rotates rows and columns p and q of A through the angle θ so that the (p,q) and (q,p) off-diagonal entries are zeroed as shown below ($*$ is any entry a_{ij}):

$$\begin{bmatrix} * & & & \cdots & & & * \\ & \ddots & & & & & \\ & & a_{pp} & \cdots & a_{pq} & & \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ & & a_{qp} & \cdots & a_{qq} & & \\ & & & & & \ddots & \\ * & & & \cdots & & & * \end{bmatrix} \rightarrow \begin{bmatrix} * & & & \cdots & & & * \\ & \ddots & & & & & \\ & & a'_{pp} & \cdots & 0 & & \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ & & 0 & \cdots & a'_{qq} & & \\ & & & & & \ddots & \\ * & & & \cdots & & & * \end{bmatrix}.$$

Based on the number of angles required for annihilating the off-diagonal elements, Jacobi methods are broadly classified as One-sided and Two-sided Jacobi methods. Two-sided Jacobi method is implemented in this work. The simplified stages of Two-sided Jacobi for solving eigenfunction problems can be described as:

1. Choose an index pair (p,q) from matrix A such that $1 < p < q < n$.
2. Calculate a (c,s) pair for the rotation matrix J such that the 2×2 subproblem matrix is diagonal:

$$\begin{bmatrix} a'_{pp} & a'_{pq} \\ a'_{qp} & a'_{qq} \end{bmatrix} = \begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix}^T \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} \tag{2.10}$$

3. Repeat by selecting the next (p,q) pairs. The algorithm fetches the next index pair (p, q) such for a sequence of N jacobi rotations, called sweep. The matrix A is overwritten at each step $A \leftarrow J^T\, A\, J$.

The 2nd stage involves solving for off-diagonal elements and equating to zero (a'_{pq} and a'_{qp} are equal to zero):

$$a_{pp}.sin\theta_1 cos\theta_2 + a_{qp}.cos\theta_1 cos\theta_2 - a_{pq}.sin\theta_1 sin\theta_2 + a_{qq}.cos\theta_1 sin\theta_2 = 0, \tag{2.11}$$

$$a_{pp}.cos\theta_1 sin\theta_2 - a_{qp}.sin\theta_1 sin\theta_2 - a_{pq}.cos\theta_1 cos\theta_2 + a_{qq}.sin\theta_1 cos\theta_2 = 0. \tag{2.12}$$

Adding and subtracting Equations 2.11 and 2.12, gives the required Jacobi rotation angles:

$$tan(\theta_1 + \theta_2) = \frac{a_{pq} + a_{qp}}{a_{qq} - a_{pp}}; tan(\theta_1 - \theta_2) = \frac{a_{pq} - a_{qp}}{a_{qq} + a_{pp}}. \tag{2.13}$$

θ_1 and θ_2 are called half angles. The goal is to diagonalize matrix A in terms of θ_1 and θ_2. Equation 2.10 can be re-written as:

$$\begin{bmatrix} a'_{pp} & 0 \\ 0 & a'_{qq} \end{bmatrix} = \begin{bmatrix} cos\theta_1 & -sin\theta_1 \\ sin\theta_1 & cos\theta_1 \end{bmatrix} \begin{bmatrix} a_{pp}.cos\theta_2 - a_{pq}.sin\theta_2 & a_{pp}.sin\theta_2 + a_{pq}.cos\theta_2 \\ a_{qp}.cos\theta_2 - a_{qq}.sin\theta_2 & a_{qp}.sin\theta_2 + a_{qq}.cos\theta_2 \end{bmatrix} \tag{2.14}$$

Replace $cos\theta_1 \Rightarrow c1$, $sin\theta_1 \Rightarrow s1$, $cos\theta_2 \Rightarrow c2$, and $sin\theta_2 \Rightarrow s2$, and carry out matrix multiplication of Equation 2.14, a'_{pp} and a'_{qq} become:

$$a'_{pp} = c1.(a_{pp}.c2 - a_{pq}.s2) - s1.(a_{qp}.c2 - a_{qq}.s2), \tag{2.15}$$

$$a'_{qq} = s1.(a_{pp}.s2 + a_{pq}.c2) + c1.(a_{qp}.s2 + a_{qq}.c2). \tag{2.16}$$

Hence, the rotation angles $cos(\theta_1+\theta_2), cos(\theta_1-\theta_2), sin(\theta_1+\theta_2), sin(\theta_1-\theta_2)$ can all be obtained by solving Equation 2.13. Thus, a solution is obtained for a'_{pp} and a'_{qq} in $\begin{bmatrix} a'_{pp} & 0 \\ 0 & a'_{qq} \end{bmatrix}$. The Two-sided Jacobi algorithm for SVD is formulated in Algorithm 4.

Algorithm 4 Two-sided Jacobi for eigenvalues and eigenvectors.

Step 1. Read the symmetric matrix A. Let $U = 1$ be a matrix of the same size as A.

Step 2. Construct matrix U and V that are identical to the unit matrix, except for $U_{pp} = U_{qq} = V_{pp} = V_{qq} = cos\theta$, $U_{pq} = V_{pq} = sin\theta$, and $U_{qp} = V_{qp} = -sin\theta$.

Step 3. Find the off-diagonal element from $A = [a_{pq}]$.

Step 4. Find the rotational angles using Equation 2.13. And hence, find U' and V'

Step 5. Then compute the matrix products $A' = V^TAU$, $Uout = U'.U$, and $Vout = V'.V$; A'_{pq} becomes zero by this operation, the other elements in rows and columns p and q are changed.

Step 6. If the number of iterations is reached or the largest absolute value of the off diagonal elements A_{pq} is larger than a threshold, repeat the process from step no. 3 with A' instead of A, $Uout$ instead of U, and $Vout$ instead of V. Upon convergence, A' contains the eigenvalues, and $Uout$ and $Vout$ contain the eigenvectors.

Algorithm 4 concludes with singular value decomposition of the symmetric matrix A.

2.6.4 Spectral Clustering Example

For illustration, consider a graph G with 5 vertices, the similarity matrix for the graph is found using some criterion to be for example:

$$W = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

An entry in the matrix W, w_{ij} represents whether node i and j share an edge. The graph corresponding to matrix W is presented below: It can be observed that for example that node 1 shares 3 edges with nodes 2,3, and 5, hence, the entries (1,2),(1,3),(1,5) are ones in the matrix W. In addition, the diagonal entries of W are zero.
The degree matrix D discussed earlier represents the sum of weights for each node, i.e.,

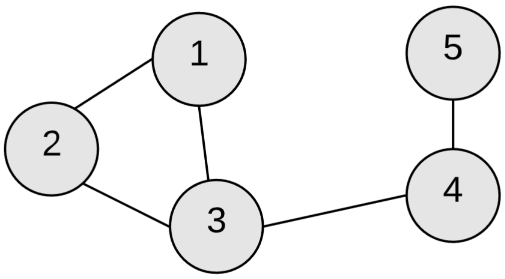


Figure 2.14: Similarity graph corresponding to matrix W.

number of connections each node has. In this example the degree matrix D becomes:

$$D = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The degree of node $ij = 11$ is 2 in this case.
The Laplacian matrix L can be found by $L = D - W$:

$$L = \begin{pmatrix} 2 & -1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$

The diagonal of L represents the degrees of the nodes while off the diagonal are binary entries of 0 or -1; 0 means that a pair of given nodes is not connected and -1 that the pair of nodes is connected. Next step is to find the eigenvalues of the matrix L.
First, we compute LL^T and L^TL. Using Corollary 1, we know that $LL^T = L^TL = L^2$, then L^2 is:

$$L = \begin{pmatrix} 6 & -3 & -4 & 1 & 0 \\ -3 & 6 & -4 & 1 & 0 \\ -4 & -4 & 12 & -5 & 1 \\ 1 & 1 & -5 & 6 & -3 \\ 0 & 0 & 1 & -3 & 2 \end{pmatrix}$$

We can determine the eigenvalues of the matrix L^2 using the roots of the characteristic equation $det(L^2 - \lambda I_m) = 0$.

$$\det\left(\begin{pmatrix} 6 & -3 & -4 & 1 & 0 \\ -3 & 6 & -4 & 1 & 0 \\ -4 & -4 & 12 & -5 & 1 \\ 1 & 1 & -5 & 6 & -3 \\ 0 & 0 & 1 & -3 & 2 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}\right) : \quad -\lambda^5 + 32\lambda^4 - 306\lambda^3 + 916\lambda^2 - 225\lambda$$

$$= -\lambda(\lambda^4 - 32\lambda^3 + 306\lambda^2 - 916\lambda + 225) = -\lambda(\lambda - 9)(\lambda^3 - 23\lambda^2 + 99\lambda - 25) = 0.$$

Solving the last equation gives the sorted eigenvalues of L^2:

$$\begin{pmatrix} \lambda_1 = 0 \\ \lambda_2 = 0.269 \\ \lambda_3 = 5.341 \\ \lambda_4 = 9 \\ \lambda_5 = 17.390 \end{pmatrix}$$

Hence, the singular values (eigenvalues of L) are $\sigma_i = \sqrt{\lambda_i}$ sorted:

$$\begin{pmatrix} \sigma_1 = 0 \\ \sigma_2 = 0.5188 \\ \sigma_3 = 2.311 \\ \lambda_4 = 3 \\ \lambda_5 = 4.17 \end{pmatrix}$$

We find that the first eigenvalue of L, σ_1, is 0 as expected. The eigenvalues are sorted to find the 2nd smallest eigenvalue, $\lambda_2 = 0.5188$ where the corresponding vector is:

$$\begin{pmatrix} -0.59696 \\ -0.59696 \\ -0.28725 \\ 0.48119 \\ 1 \end{pmatrix}$$

The nodes on the graph can be labelled with 0 or 1 classes according to each value in the eigenvector (positive or negative). It gives us information about which side of the cut a node belongs to.

Extensions to multiple partitions are possible via recursive bipartitioning or through the use of multiple eigenvectors. In this work, we make use of multiple eigenvectors to find partitions of more than just 2 clusters. This is done by spectral embedding. Spectral embedding aims to represent a graph in some Euclidean space of low dimension, say $\mathbb{R}^{N_E}$ where $N_E \ll n$. Hence, each node $i \in V$ is represented by vector $e_i \in \mathbb{R}^E$. The structure of the graph must be encoded in its representation $e_1, e_2, \ldots, e_n$ such that two close nodes i, j in the graph G correspond to two close vectors e_i, e_j in the embedding space.
The importance of spectral embedding in spectral clustering can be shown in Figure 2.15. Mapping the nodes of the graph into higher dimensions shows the distances between the nodes. Eigenvectors of the Laplacian matrix provide an embedding of the data based on similarity, i.e, groups the closer nodes together. To find such an embedding for the normalized cut problem, [37] computed the $n \times N_E$ matrix of the leading eigenvectors V and the $N_E \times N_E$ diagonal matrix of eigenvalues Λ of the system:

$$D^{-\frac{1}{2}}(D - W)D^{-\frac{1}{2}}V = V\Lambda.$$

The ith embedding representation of the jth pixel is then given by

$$E_{ij} = \frac{V_{i+1,j}}{\sqrt{D_{jj}}}, \quad i = 1, \ldots, N_E, j = 1, \ldots, N, \tag{2.17}$$

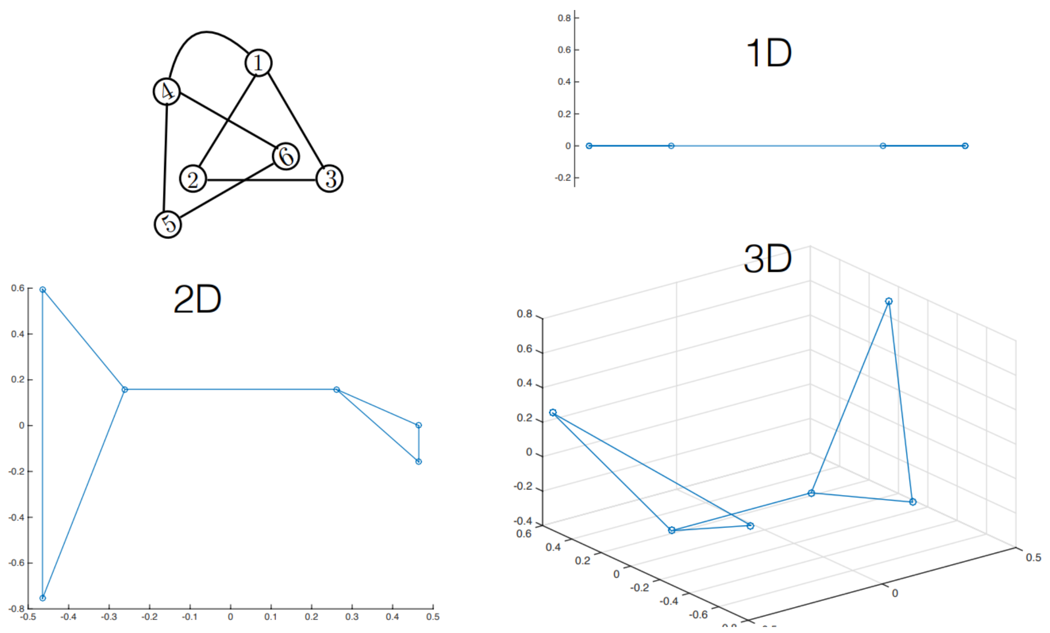


Figure 2.15: Spectral embedding example.

where the eigenvectors have been sorted in ascending order by eigenvalue. Thus, each pixel is associated with a column of E and the final partitioning is accomplished by clustering the columns for each individual pixel. This corresponds to the normalization of the first k eigenvectors that takes place in 3 before using k-means to discover groups of pixels in this embedding space.

2.6.5 Nyström Extension

Solving eigensystem problems with large matrices such as the ones expected to be constructed for hyperspectral images can be high computationally expensive. The Nyström extension is a technique for finding numerical approximations to eigenfunction problems formulated out of large matrices such as the similarity matrix of a hyperspectral image. The method finds the numerical solution of an integral equation by replacing the integral with a representative weighted sum. [12] introduced making use of Nyström extension for spectral clustering of hyperspectral images to obtain cluster results efficiently by finding approximations for the leading eigenvectors of the Laplacian matrix and hence, a solution of the eigenfunction problem.
This is done by taking a sample pixels from the original input data set, and get the affinity matrix for it. Then, we find eigenvalue decomposition of the obtained affinity matrix including the eigenvectors V. Nyström method extends the eigenvectors obtained to V_e such that V_e is an approximation of the original eigenvectors.
The goal in the coming discussion is to approximate the similarity matrix by considering it to be an eigenfunction problem and then extend the result to the normalized cut concept discussed in section 2.6.

In the context of hyperspectral images, the affinity matrix considers both the reflectance value of the pixels and their spatial location in order to define the similarity between two samples x_i and x_j. Hence, equation 2.3 of section 2.6 can be redefined as:

$$W_{ij} = \exp\frac{-||l_i - l_j||_2^2}{2\sigma_l^2}.\exp\frac{-||x_i - x_j||_2^2}{2\sigma_x^2}, \qquad i,j = 1,2,\dots,n, \tag{2.18}$$

where l_i and l_j are the spatial locations of the HSI's pixels and σ_l and σ_x are the bandwidth of neighbouring pixels. σ_l and σ_x are sensitive to different hyperspectral images. According to [38], these parameters can be related to l_i and l_j and x_i and x_j, and we can define $\bar{\sigma_l}$ and $\bar{\sigma_x}$ as

$$\bar{\sigma_l} = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}||l_i - l_j||_2^2, \bar{\sigma_x} = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}||x_i - x_j||_2^2. \tag{2.19}$$

Substituting back to Equation 2.8, it becomes:

$$W_{ij} = \exp\frac{-||l_i - l_j||_2^2}{2\alpha\bar{\sigma_l}^2}.\exp\frac{-||x_i - x_j||_2^2}{2\alpha\bar{\sigma_x}^2}, \qquad i,j = 1,2,\dots,n, \tag{2.20}$$

where α is a parameter or factor to control the number of neighbors for each node in the affinity matrix.

Let A be a matrix defined by Equation 2.18, then A is an affinity matrix of m chosen samples. The similarity matrix of the remaining $n-m$ samples and the chosen samples are denoted as B. In addition, let C denote the affinity matrix for the remaining samples alone. The affinity matrix W of equation 2.8 can be rewritten as

$$W = \begin{bmatrix} A & B \\ B^T & C \end{bmatrix}, \tag{2.21}$$

where B^T is the transpose of matrix B. Hence, $A \in \mathbb{R}^{m\times m}$, $B \in \mathbb{R}^{m\times n}$, and $C \in \mathbb{R}^{n\times n}$. The affinity matrix C can be extremely large for HSI images. According to Nyström extension, if we know A and B we can find an approximation for matrix C such that $C = B^T A^{-1} B$. In other words, given only m rows of the matrix W, we can find the values of the entire $n \times n$ matrix W. In the following discussion, we establish the explanation of Nyström extension method in the steps similiar to [39].

Eigenfunction problems in which the Nyström extension is used are in the form

$$\int_a^b W(x,y)\phi(y)\,\mathrm{d}y = \lambda\phi(x).$$

According to [39], we can approximate this integral equation by evaluating it at a set of evenly spaced points $\epsilon_1, \epsilon_2, \dots, \epsilon_n$ on the interval $[a,b]$ and employing an approximation (called, quadrature rule; more information can be found in Appendix 2) of the definite integral of the function $W(x,y)\phi(y)$ such that it becomes:

$$\frac{(b-a)}{n}\sum_{j=1}^{n} W(x,\epsilon_j)\hat{\phi}(\epsilon_j)=\lambda\hat{\phi}(x), \tag{2.22}$$

where $\hat{\phi}(x)$ is an approximation to the true $\phi(x)$. Set $x=\epsilon_i$ and equation 2.22 becomes

$$\frac{(b-a)}{n}\sum_{j=1}^{n} W(\epsilon_i,\epsilon_j)\hat{\phi}(\epsilon_j)=\lambda\hat{\phi}(\epsilon_i) \qquad \forall i\in\{1\ldots n\}. \tag{2.23}$$

Let $[a,b]$ be $[0,1]$, we structure the system 2.23 as a matrix eigenvalue problem:

$$A\widehat{\Phi}=n\widehat{\Phi}\Lambda$$

where $A_{ij}=W(\epsilon_i,\epsilon_j)$ and $\widehat{\Phi}=[\hat{\phi}_1,\hat{\phi}_2,\ldots,\hat{\phi}_n]$ are the n eigenvectors of A corresponding to $\lambda_1,\lambda_2,\ldots,\lambda_n$ placed on the diagonals of the matrix Λ. Substituting back to equation 2.22 yields the Nyström extension for each eigenvector $\hat{\phi}_i$:

$$\hat{\phi}_i(x)=\frac{1}{n\lambda_i}\sum_{j=1}^{n} W(x,\epsilon_j)\hat{\phi}_i(\epsilon_j). \tag{2.24}$$

This expression allows us to extend an eigenvector computed for a set of sample points to an arbitrary point x using $W(.,\epsilon_j)$ as the interpolation weights.

Considering A again to be the affinity matrix present in matrix W of Equation 2.21. It has been established in Section 2.6 that the Laplacian matrix is a positive semi-definite matrix. Recall the eigenvalue decomposition analysis of A from Section 2.6.3:

$$A=U\Lambda U^T$$

which is called the diagonalization of A.
Going back to the Nyström extension for $\hat{\phi}_i$ in 2.24, since matrix B represents the weights between the sample points and the remaining points, then, the matrix form of the Nyström extension is $B^TU\Lambda^{-1}$ where B^T corresponds to $W(\epsilon_j,.)$, the columns of U, the eigenvector, corresponds to $\hat{\phi}_i(\epsilon_j)$s, and Λ^{-1} corresponds to $1/\lambda_i$s. Recall the matrix W of 2.21, letting $\bar{U}$ denote the approximate eigenvectors of W, then

$$\bar{U}=\begin{bmatrix} U \\ B^TU\Lambda^{-1}\end{bmatrix}, \tag{2.25}$$

since U is the eigenvectors of A and $B^TU\Lambda^{-1}$ is the Nyström extension matrix form. Furthermore, we can approximate W, which we denote, $\widehat{W}$, such that

$$\widehat{W} = \bar{U}\Lambda\bar{U}^T \tag{2.26}$$

$$= \begin{bmatrix} U \\ B^T U \Lambda^{-1} \end{bmatrix} \Lambda \begin{bmatrix} U^T & \Lambda^{-1} U^T B \end{bmatrix} \tag{2.27}$$

$$= \begin{bmatrix} U\Lambda U^T & B \\ B^T & B^T A^{-1} B \end{bmatrix} \tag{2.28}$$

$$= \begin{bmatrix} A & B \\ B^T & B^T A^{-1} B \end{bmatrix} \tag{2.29}$$

$$= \begin{bmatrix} A \\ B \end{bmatrix} A^{-1} \begin{bmatrix} A & B \end{bmatrix} \tag{2.30}$$

$$\tag{2.31}$$

since $UU^T = I = \Lambda\Lambda^{-1}$. Hence, $C = B^T A^{-1} B$ according to the approximation of the matrix W using Nyström extension.

We are left to extend the above approximation to the normalized cut defined earlier in section 2.6. We know that by definition, $\widehat{d}$ represent the row sum of matrix W, hence,

$$\widehat{d} = \widehat{W}\mathbf{1} = \begin{bmatrix} A\mathbf{1}_m + B\mathbf{1}_n \\ B^T\mathbf{1}_m + B^T A^{-1} B\mathbf{1}_n \end{bmatrix}, \tag{2.32}$$

where $A\mathbf{1}_m$ and $B\mathbf{1}_n$ are the row sum of matrices A and B and $B^T\mathbf{1}_m$ is the column sum of matrix B. $\mathbf{1}_m$ is a vector row of **1**s of size m.
Having $\hat{d}$, we can approximate the blocks of $D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}}$, (similar to the embedding done in Equation 2.17) such that the matrix A and B can be formulated as:

$$A_{ij} = \frac{A_{ij}}{\sqrt{\widehat{d_i}\widehat{d_j}}} \tag{2.33}$$

$$B_{ij} = \frac{B_{ij}}{\sqrt{\widehat{d_i}\widehat{d}_{j+m}}} \tag{2.34}$$

and we can get the normalized affinity matrix $D^{-\frac{1}{2}}\widehat{W}D^{-\frac{1}{2}}$, hence,

$$D^{-\frac{1}{2}}\widehat{W}D^{-\frac{1}{2}} = \begin{bmatrix} A \\ B \end{bmatrix} A^{-1} \begin{bmatrix} A & B \end{bmatrix}. \tag{2.35}$$

In summary, Algorithm 3 becomes

Algorithm 5 Spectral Clustering with Nyström extension method [37].

```
Input: dataset $S = \{s_1, s_2, \dots, s_n\}$, number of clusters $k$
Output: $k$-clustering of $S$
$W_{ij} = \exp \frac{-||l_i - l_j||_2^2}{2\sigma_l^2} . \exp \frac{-||x_i - x_j||_2^2}{2\sigma_x^2}, \qquad i, j = 1, 2, \dots, n.$
$D_{ij} = d_{ii} = \sum_j W_{ij}$ .
Calculate matrices $A$ and $B$ using Equations 2.33 and 2.34.
Compute the normalized Laplacian
    $$L = D^{-\frac{1}{2}} \widehat{W} D^{-\frac{1}{2}} = \begin{bmatrix} A \\ B \end{bmatrix} A^{-1} \begin{bmatrix} A & B \end{bmatrix}.$$
Find $v_1, v_2, \dots, v_k = SmallestEigenVectors(L, k)$.
Let $V \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $v_1, v_2, \dots, v_k$ as columns.
Find the embedding $E_{ij} = \frac{V_{i+1,j}}{\sqrt{Djj}}, \quad i, j = 1, \dots, k.$
For $i = 1, \dots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of $E$.
Cluster the points $(y_i)_{i=1,,\dots,n}$ in $\mathbb{R}^k$ with the k-means algorithm into clusters $C_1, \dots, C_k$.
```

2.7 Overview of Zynq-7000 Functional Blocks

Zynq-7000 is a family of Xilinx System-on-Chip (SoC) products which offers the flexibility and scalability of an FPGA, while providing performance, power, and ease of use. The SoC architecture enables implementation of custom logic in the processing logic (PL) and custom software in the processing system (PS), and thus, enables implementations of HW/SW co-designs. The overview of the Zynq architecture is shown in Figure 2.16, showing the functional blocks of both the PL and the PS.
The processing system comprises four major blocks:

- Application Processor Unit (APU) which includes 2 ARM Cortex-A9 cores.
- Memory Interfaces: A dynamic memory controller and static memory interface modules.
- I/O Peripherals (IOP) which contains the data communication peripherals.
- Central Interconnect through which the APU, memory interface unit, and the IOP are all connected to each other and to the PL.

Making use of the four main blocks of the PS along with a special PS-PL interface allows high performance communication between PS and PL cores on a single chip, which further makes HW/SW co-design implementations possible. The processing logic cores used in this thesis include DSP blocks, BRAM blocks, and DMA core. The PS-PL interface is conducted by AXI protocols explained below in this section.

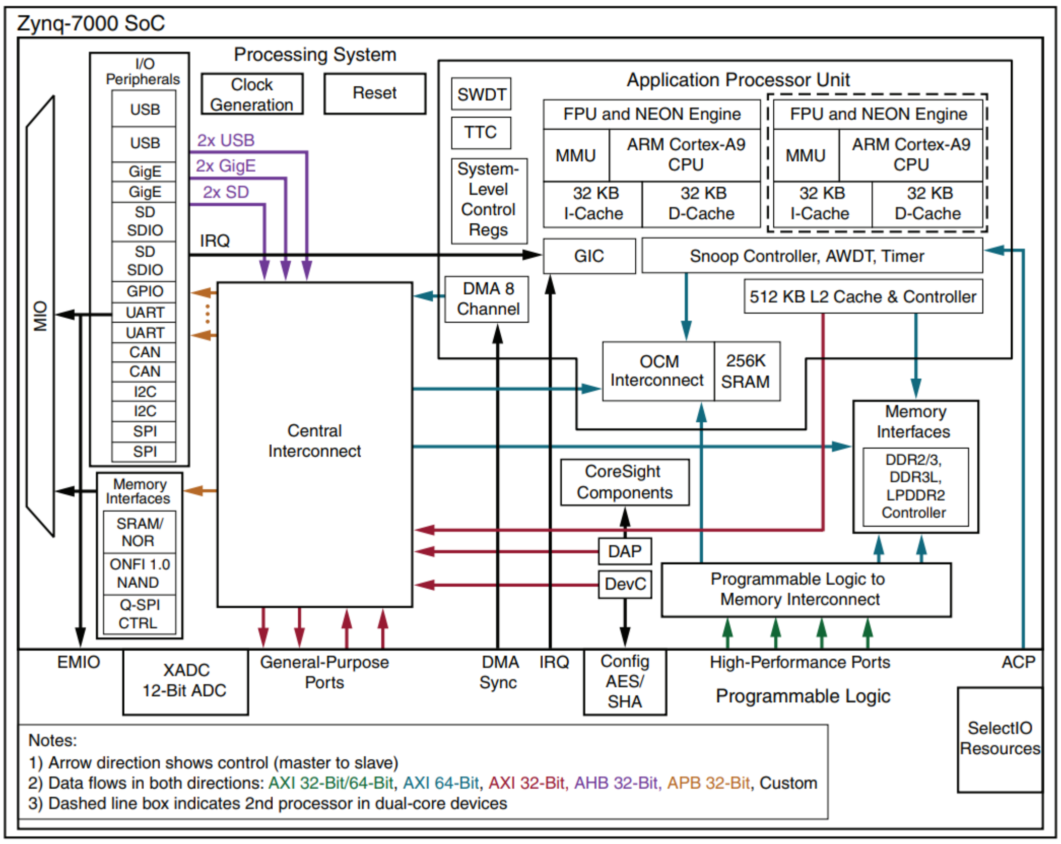


Figure 2.16: Architectural Overview of Zynq-7000 [40].

2.7.1 DSP blocks on Zynq

Digital signal processing (DSP) applications typically require a large number of mathematical operations to be performed quickly and repeatedly on a series of data samples. Such applications have constraints on latency, and dedicated hardware chips are designed to meet these constraints. Zynq SoC has 220 dedicated, full-custom, low-power DSP slices, combining high speed with small size while retaining system design flexibility [41]. A DSP slice found on Zynq SoC is depicted in Figure 2.17. Each DSP slice contains a pre-adder, 25×18 bit multiplier, and a 48-bit accumulator, all capable of operating up to 741 MHz. In addition, three pipeline data registers are provided to run at full speed.

The DSP48E1 multiplier has asymmetric inputs and accepts two's complement operands of 25 and 18-bits. The multiplier stage produces a 43-bit two's complement result in the form of two partial products. These partial products are sign-extended to 48 bits in the multiplexer and fed into three-input adder for final summation. This results in a 43-bit multiplication output, which has been sign-extended to 48 bits. The C input port allows the formation of many 3-input mathematical functions, such as 3-input addition or 2-input multiplication with an addition. The second stage adder/subtracter accepts three 48-bit two's complement operands and produces a 48-bit two's complement result. In addition, the DSP48E1 slice is the only FPGA architecture that supports pattern detection which is used for convergent rounding and overflow/underflow detection.

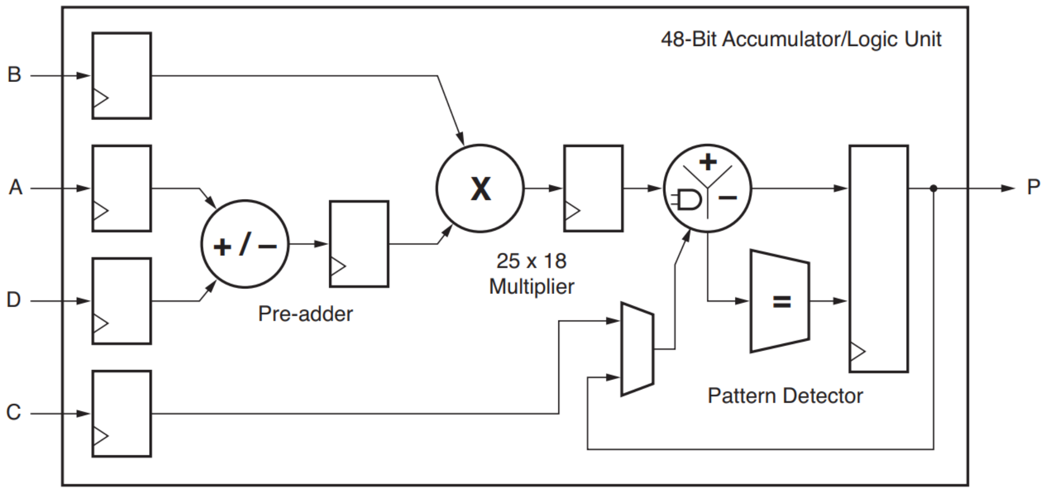


Figure 2.17: Basic DSP48E1 Slice Functionality [41].

2.7.2 AXI Protocols

All devices in a Zynq system communicate each other based on a device interface standard developed by ARM, called AXI (Advanced eXtended Interface). There are three types of AXI bus interfaces: AXI (full), AXI-lite and AXI-stream. As Zynq SoC uses AMBA 4.0 (Advanced Microcontroller Bus Architecture) released in 2010 [40], the focus is placed on that interface version which consequently includes AXI-4, AXI-4 Stream, and AXI4-Lite. Figure 2.18 illustrates the AXI interconnects that will be used in this work. The processor and DDR memory controller are contained within the Zynq PS. The AXI DMA and PL Block are implemented in the Zynq PL. The AXI-lite bus allows the processor to communicate with the AXI DMA to setup, initiate and monitor data transfers. The AXI_MM2S and AXI_S2MM are memory-mapped AXI4-Stream buses and provide the DMA access to the DDR memory. AXI4-Lite and AXI4-Stream interconnects are discussed below.

AXI4-Lite Interface

The AXI4-Lite interface developed by ARM [42], consists of five channels: read address, read data, write address, write data, and write response. All five transaction channels use a VALID/READY handshake process to transfer address, data, and control information. Each channel is independent from each other and has its own couple of VALID/READY handshake signals. The information source drives the VALID signal to inform the destination entity that the payload on the channel is valid and can be read. Similarly, the READY signal is generated by the information destination to indicate that it can accept that information. The handshake completes if both VALID and READY signals in a channel are asserted during a rising clock edge.

Communication over an AXI4-Lite bus incorporates two interconnects, a master and a slave. An AXI4-Lite read and an AXI4-Lite write using the five channels is shown in Figure 2.19. To start a read transaction, the master has to provide data address and data length on the Read address channel. After the usual VALID/READY handshake,

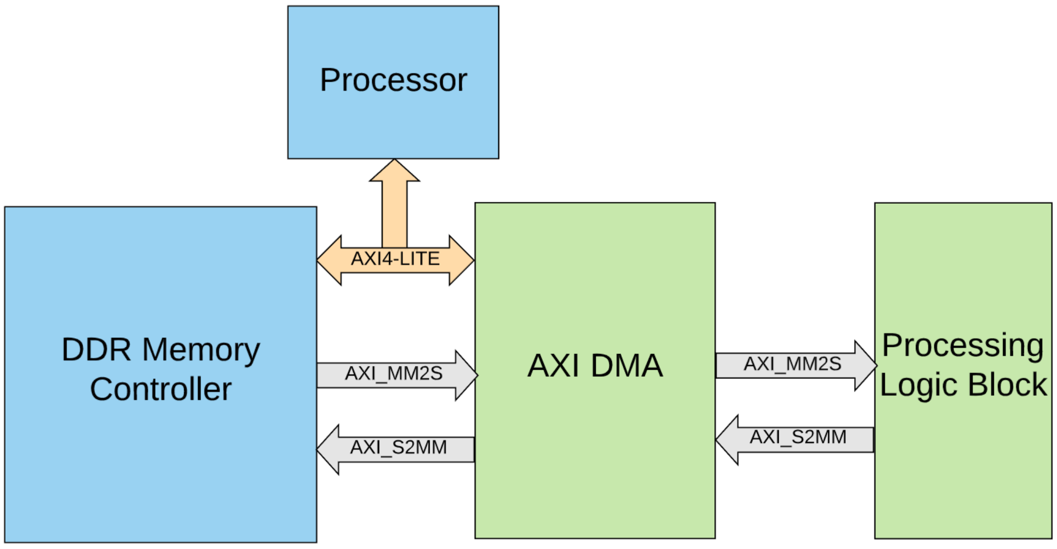
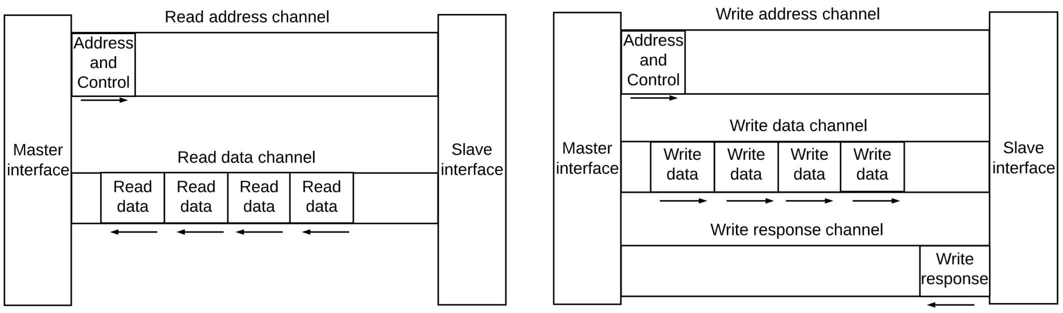


Figure 2.18: Block diagram of AXI interconnects connecting the processor and DMA (in PS) to AXI DMA and a processing logic block (in PL).



(a) AXI4-Lite read channels. (b) AXI4-Lite write channels.

Figure 2.19: AXI4-Lite interface [42].

the slave has to provide the data corresponding to the specified address on the Read data channel as shown in Figure 2.19(a). Similarly, writing transaction is started by the master by sending an address on the write address channel and corresponding data on the write data channel. After the transaction completion on both channels, the slave has to send back to the master the status of the data write over the Write response channel as depicted in Figure 2.19(b).

AXI4-Stream Interface

AXI4-Stream by ARM [43], is used as a standard interface to connect components that wish to exchange a continuous stream of data. The interface can be used to transport data streams of arbitrary width in hardware. Most usually 32-bit bus width is used to be connected to the DMA core which performs memory-mapped to stream (MM2S) conversion. The interface handshake protocol is similar to AXI4-Lite but uses an additional handshake signal (TLAST) and is depicted in Figure 2.20. The interfaces of this type do not use data addresses.

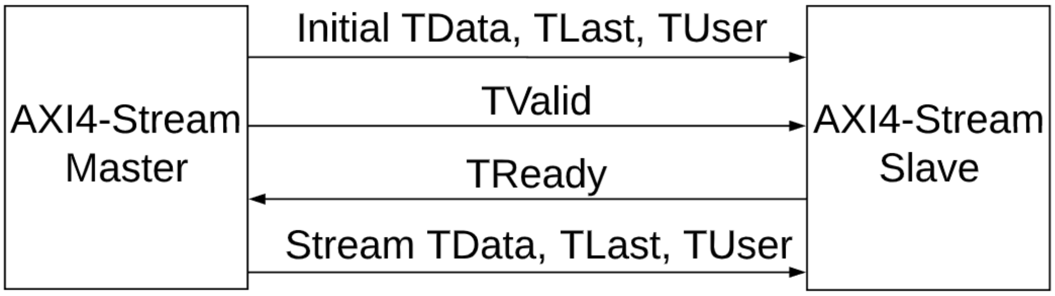


Figure 2.20: AXI4-Stream Interface.

In AXI4-Stream, TDATA width of bits is transferred per clock cycle. The transfer is started once the master signals TVALID and the slave responds with TREADY which represents that it is ready to receive data. TLAST signals the last byte of the stream. TUSER is an optional signal which is a user defined sideband information that can be transmitted alongside the data stream [43]. In addition, TID and TDEST are two optional signals where TID usually identifies which master the stream originated from, and TDEST is used to route the stream to the appropriate destination.

2.7.3 AXI DMA

Direct Memory Access (DMA) cores are used to allow direct memory access to peripherals with minimal CPU intervention. In the case of a Zynq system, AXI DMA provided by Xilinx [44], is instantiated in PL and it is used to transfer data from DDR memory to the FPGA cores with AXI-stream interfaces as shown in Figure 2.18. The AXIS_MM2S and AXIS_S2MM are AXI4-streaming buses, which source and sink a continuous stream of data without addresses. In this work, we use the DMA to transfer data from memory to an PL block and back to the memory. In principle, the PL block could be any kind of data producer/consumer. In our case, it is the processing logic block of the intended HW/SW co-design.

2.7.4 Block RAM

One of the main challenges in hardware implementation of matrix operations is storing the data. As the real symmetric matrix is quite huge, it needs to be stored in memory and filled into and pulled from the memory to perform operations on it. The Xilinx block RAM (BRAM) is a dual-port RAM module instantiated into the FPGA fabric to provide on-chip storage for a relatively large set of data. Zynq-7000 features 36Kb BRAM dual-port units which allow for parallel, same-clock-cycle access to different locations [45]. In this work, the BRAM is modified such that it has 1 write and 2 read ports for a vector of data. In other words, there are two read pointers to data addresses which can be decoded into 2 separate or the same data row. Figure 2.21 shows the design of a dual-port vector BRAM with write enable (we) used in this work. The BRAM is defined by its size (BRAM_ADDR_WIDTH) and width of each row (BRAM_DATA_WIDTH). For a write operation, din represents the input data to be written on the RAM address specified by the row (w_addr) on the BRAM. Each row is associated with a write enable signal (we).

On the other hand, on each rising clock edge, the specified addresses, r_addr_1 and r_addr_2, are read and given to dout_1 and dout_2, respectively.

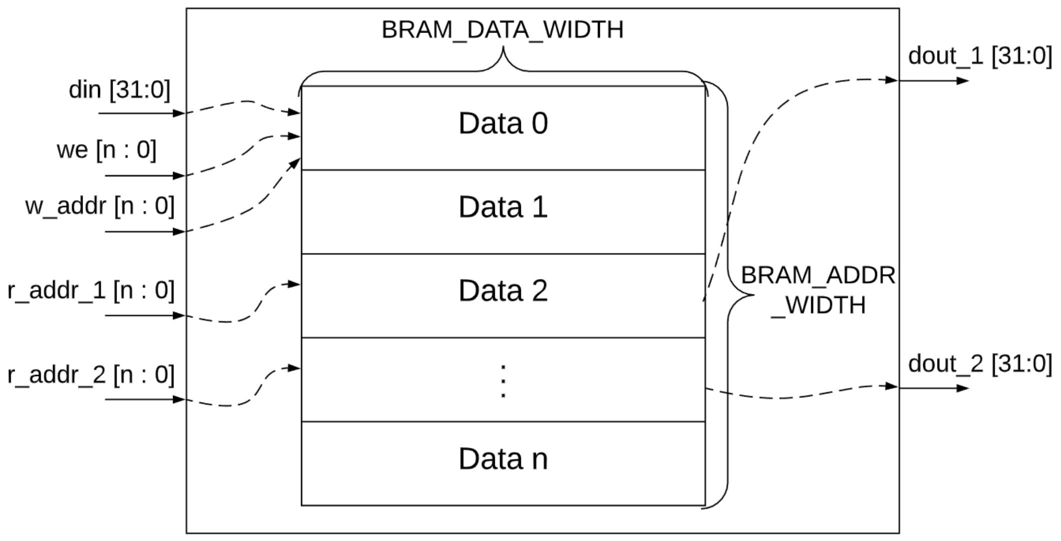


Figure 2.21: Vector storage using BRAM.

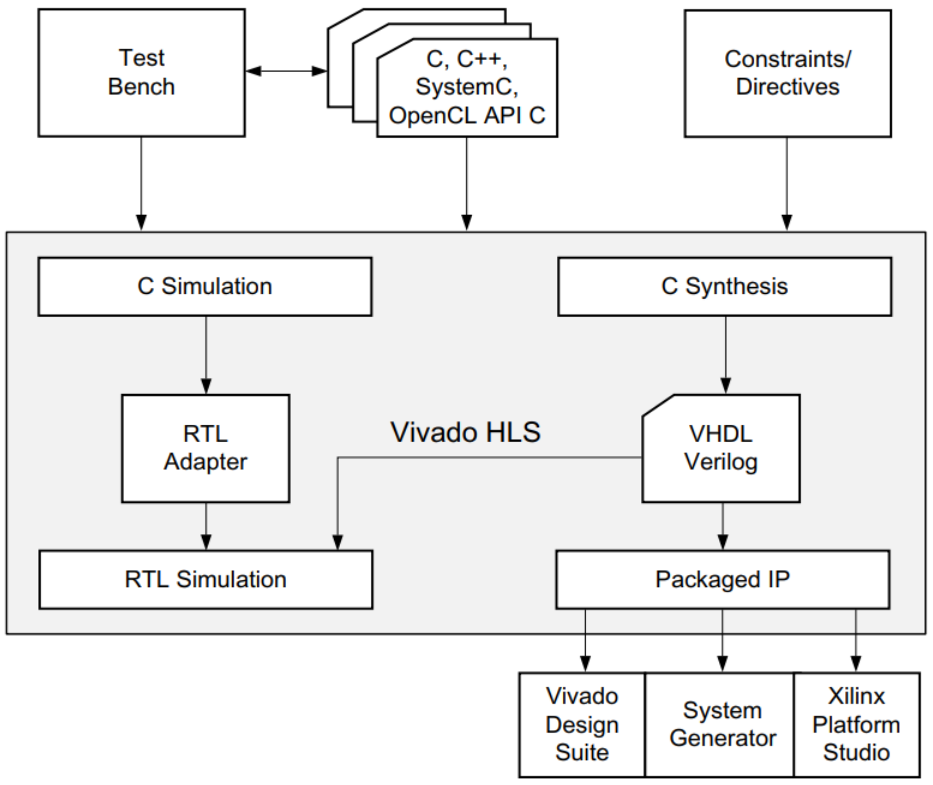


Figure 2.22: Vivado HLS Design Flow [46].

2.8 Vivado HLS

The Vivado High-Level Synthesis (HLS) is part of Vivado Design Suite produced by Xilinx [46]. This HLS tool converts algorithmic descriptions written in C-based design flow into hardware descriptions (RTL). The RTL implementations can be further synthesized for FPGA platforms on Xilinx products, such as, Zynq-7000.

Figure 2.22 shows the Vivado HLS design flow. The user starts the design flow by supplying a functional C code design and a C testbench design. Along with the C code, the user may also specify the directives which direct the HLS to implement a specific behaviour or hardware optimization. The functional verification of C code is done by C simulation.

Next, C-to-RTL HLS is done by C Synthesis along with the provided directives and constraints used to define and refine the RTL implementation. Until this stage, the user is provided with an estimation of the list of hardware resources. Once the RTL is generated using HLS, C/RTL cosimulation verifies the RTL output using the same testbench, however, the HW testbench is also generated by the HLS tool. The user is provided with a report to observe the HW and the SW results and a functional comparison can be deduced. Finally, the implemented RTL can be exported to hardware for use on different platforms, including Vivado Design Suite, by packing it into an intellectual property (IP) block.

Table 2.1: Vivado HLS optimization directives [46].

Directive	**Description**
#pragma HLS interface	Specify RTL I/O port types for the top function level. For example, AXI_Stream I/O, bram, etc.
#pragma HLS inline	Removes a function as a separate entity in the hierarchy. After inlining, the function is dissolved into the calling function and no longer appears as a separate level of hierarchy in the RTL.
#pragma HLS unroll	Unroll loops to create multiple independent operations rather than a single collection of operations.
#pragma HLS pipeline	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.

In this thesis, Vivado HLS is used to implement the singular value decomposition (SVD) to obtain the eigenvalue and eigenvector decomposition of the Laplacian matrix for the Spectral Clustering solution. The HLS SVD implementation is described in Chapter 5. Throughout the implementation, frequently used Vivado HLS directives (pragmas) are utilized and described in Table 2.1. A user can make use of the directives to transform an initial RTL design into both a low-area and high-throughput implementation. Thus, optimizing the design. In addition, optimization directives can be used to define the desired I/O interfaces to the HLS core.

Chapter 3

Software Implementation

In this chapter, the software implementations of both methods developed in this thesis are discussed. First, the implementation of a new segment-based clustering is detailed. Then, the application of a spectral clustering based method is explained.

3.1 Segment-based Clustering using BPT

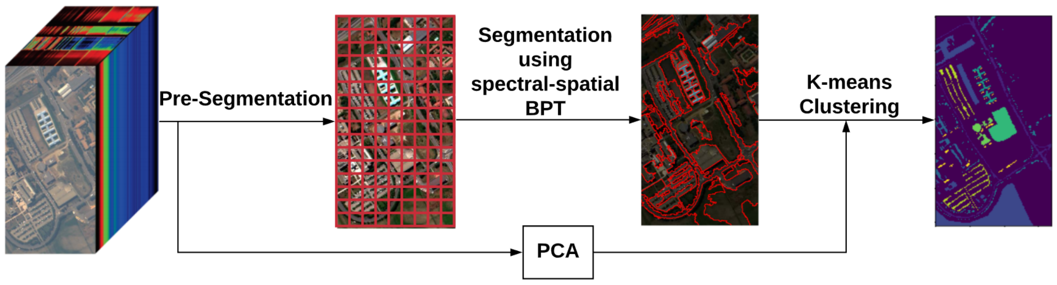


Figure 3.1: Flowchart of CLUS-BPT.

The method implemented in the specialization project [6] is further improved by adding to it a binary partition tree pruning algorithm. The framework developed in this thesis, termed as CLUS-BPT (Clustering using Binary Partition Trees), is based on a three-stage scheme: (i) pre-segmentation, (ii) segmentation, and (iii) PCA and k-means clustering. The framework is an implementation of different components described by [9] and [30]. The way the components are connected and used together is the novelty behind this framework.
Figure 3.1 depicts the working procedure of the proposed scheme. As shown, CLUS-BPT starts by making use of an initial partition to form the segmentation map (spatial discrimination), which is then used for BPT construction along which region modelling and region merging is included. In the next step, principal component analysis (PCA) (explained in Appendix A) is applied to the original HSI cube to obtain the maximum data variance direction (1st PCA component) and the result is embedded with the final segmentation map produced by BPT. Finally, the filtering algorithm-based k-means clustering method is carried over the refined segments to output the cluster map as shown in Figure 3.1.

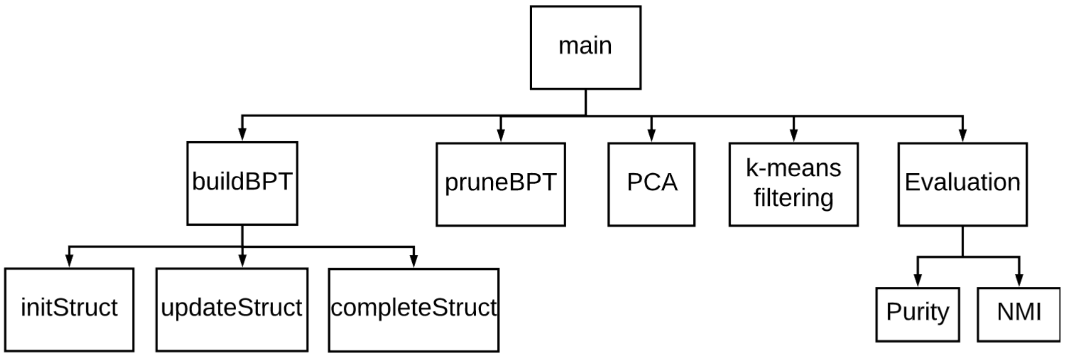


Figure 3.2: Overall software modules hierarchy.

The software implementation is carried out using MATLAB. The hyperspectral cube is loaded with dimensions of $(x \times y \times z)$, where x and y are the spatial dimensions (image size) and z is the number of bands. Throughout this implementation, the data structure maintained is a binary tree data structure. Each parent node has two children nodes as depicted in Figure 2.7. The software is designed and implemented to be able to provide a structural overview and to easily be able to determine the location of each module. Figure 3.2 shows the overall design of the software implementation of CLUS-BPT. At the top hierarchy level is the main function which contains the start of the software. The main function also calls the other sub-functions as shown in the figure. Furthermore, the main function accepts user defined parameters shown in Table 3.1.

Table 3.1: User defined parameters.

Parameter	Parameter Description	Allowed Values
File_path_1	Path to input hyperspectral data	mat datatype file & string
P_R	Number of pruned regions	Positive integer
PCA_components	Number of PCA components	Positive integer
k	Number of clusters	Positive integer
File_path_2	Path to output evaluation results	string
File_path_3	Path to output clustered labels	string

3.1.1 Pre-Segmentation

The iterative region merging algorithm for building the tree starts by making use of either the original pixels of the HSI cube or an initial region partition. The initial partition of the hyperspectral cube helps in reducing the computational time for BPT building, rather than using the original set of pixels [26]. This is because the initial partition consists of regions in which each region contains a number of pixels grouped together. Furthermore, the region merging algorithm will deal with a smaller size of input rather than the whole HSI cube.
In this work, Watershed segmentation method is utilized to obtain an initial partition of the HSI cube. The reason behind using this method is to avoid obtaining an under-segmented map (different regions with large spatial size) in which regions will not be allowed to be split later during the process. This is because the BPT will be built on

those splits and the segment sizes will further increase throughout the building process. Instead, an oversegmented map is produced which contains regions small and accurate enough to be able to be reconstructed with good accuracy in the BPT building process [26].

The implementation is based on three built-in MATLAB functions. Two of those functions are used to obtain the gradient for each spectral band, namely, *imdilate* and *imerode* functions. Figure 3.3 shows an illustration of the two functions on a hyperspectral image. Using a square structuring element of size 3, *imdilate* dilates the given image by gradually enlarging the boundaries of regions of foreground pixels. On the other hand, *imerode* erodes away the boundaries of regions of foreground pixels. Thus areas of foreground pixels shrink in size, and holes within those areas become larger. The step of subtracting the dilated image from the eroded image produces light and dark pixels which can be considered as boundaries of each region for every spectral band, called gradient values. The gradient function then gets the maximum of gradient value for each spectral band. Finally, the built-in watershed MATLAB function finds "watershed ridge lines" in the output image by treating it as a surface where light pixels represent high elevations and dark pixels represent low elevations (like high and low elevations in a mountain that is split by the watershed line).
By the end of this stage, an initial segmented map of oversegmented regions is obtained as shown in Figure 3.1.

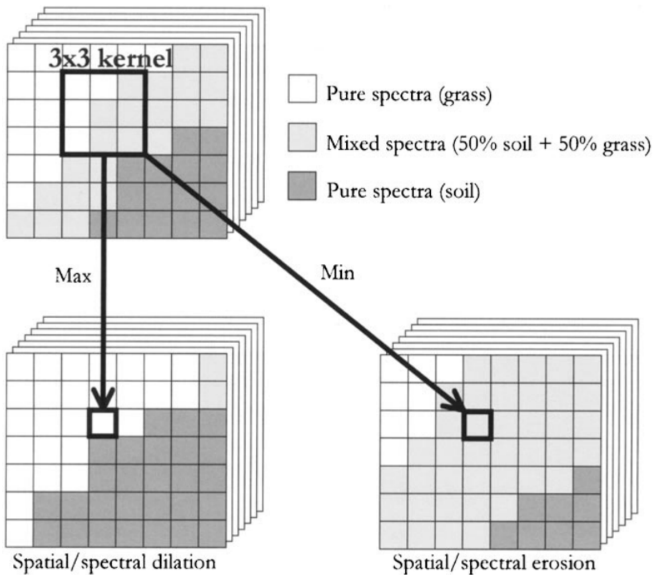


Figure 3.3: Example operations of dilation and erosion on a hyperspectral image.

3.1.2 BPT Building

In this stage, a *struct* type of data is created (called DATA). DATA has the following fields: *data_input, initsegmap, regionmodel, merging_criterion, spec_merging*.
The buildBPT function takes the DATA struct and returns the tree structure of BPT. The function starts by reshaping the data input passed through the data_input field. The

built-in MATLAB *reshape* function is used to resize the data from (x, y, z) to $(x \times y, z)$. The buildBPT function then calls three subfunctions sequentially: *initStruct, updateStruct, completeStruct*.

initStruct

This subfunction takes the initial segmentation map and hyperspectral data to create the tree data structure and initializes it. The data structure of a tree node is presented in Figure 3.4. Each node represents a region on the segmentation map.

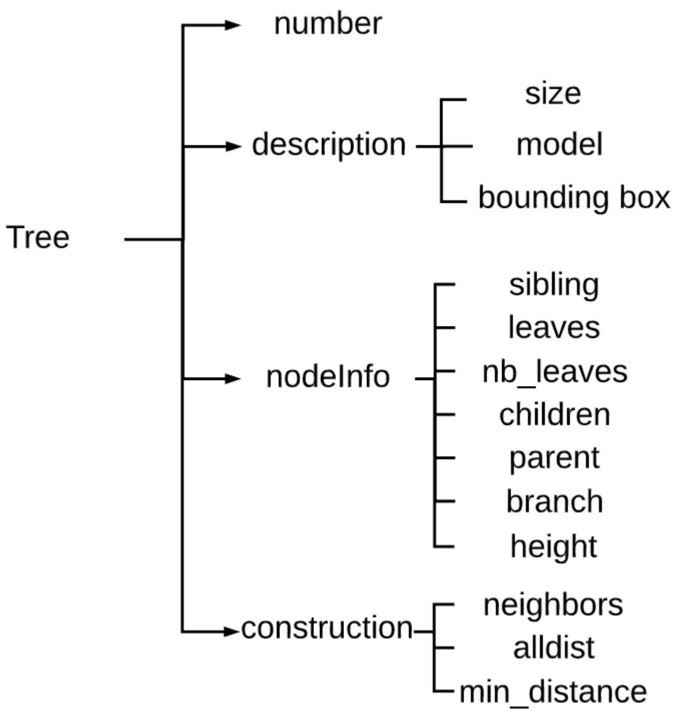


Figure 3.4: A tree node data structure.

- **number**: Unique node number. The tree size is $2 * N - 1$ where $N = \max(x, y)$.
- **definition**: This field is a struct containing the description of the region's *size*, *model* representing that region, and the region *bounding box*.
- **construction**: This is another struct of data containing the *neighbours* surrounding the current region's bounding box, the *distances* between the current node and its neighbours, and a value representing the *minimum distance*.
- **nodeInfo**: This is also a struct of information noting:
  - The closest neighbor, called *sibling*.
  - The current node may have *leaves* which represent an array of the first nodes merged at the *updateStruct* function to form the current node in the tree,
  - Number of leaves (*nb_leaves*),
  - 2 *children* and 1 *parent* form the final structure of the tree at *completeStruct*

function. An internal node can possibly be formed by a merge between one node that is a leaf, and another that is not a leaf, hence, the term *children*.

- *Branch* represents an ordered set of nodes starting from the current node up to the root of the tree to keep track of data.
- The *height* of the tree.
- The build *iteration*. This value is a counter that is maintained to keep track of the order of nodes being added to the tree. It is used by ***pruneBPT*** to prune tree nodes having a build order greater than *iteration*.

The ***model*** of the node is found by getting the pixel values (along all the z bands) of the labels corresponding to the same labels on the given initial segmentation map. The function then computes their mean (the mean of each row as in Figure 2.10) and the number of these pixel values represent the ***size*** of the region.
initStruct function then computes the ***bounding box*** for the formed model. It does that by making use of a built-in MATLAB function called
regionprops(map,'Boundingbox') which returns position and size of the smallest box containing a region inside the initial segmentation map provided by the Watershed method. The ***bounding box*** rectangle for a node is then defined as [xmin xmax ymin ymax].
The bounding box position information can be used further to identify the ***neighbors***' labels which is an array stored in the ***construction*** field. ***initStruct*** then executes the below Listing 3.1 to initialize the rest of the ***construction*** and ***nodeInfo*** fields. Lines 7 - 11 calculates the spectral angle mapper (SAM) between a tree node and all its neighbors. Lines 13 - 17 assigns the shortest distant neighbour to the ***sibling*** field.

Code listing 3.1: Initialize node structures

```
 1  for i=1:N
 2      neigh = tree(i).construction.neighbors;  % get the list of neighbors
 3      Nei = neigh(neigh>tree(i).label);
 4      % for all neighbours with labels greater than the current label
 5      % (so that we go ascending order and not repeat the process with the past nodes)
 6      D = zeros(size(Nei));
 7      for j=1:length(Nei) % calculate SAM for both ij and ji.
 8          D(j) = SAM(tree(i).descriptors.model,tree(Nei(j)).descriptors.model);
 9          ind = tree(Nei(j)).construction.neighbors == i;
10          tree(Nei(j)).construction.alldist(ind) = D(j);  % D12=D21
11      end
12      tree(i).construction.alldist(neigh>tree(i).label) = D; % write to node i
13      tree(i).construction.dist = min(tree(i).construction.alldist);
14      %find the index of the shortest distance neigbhour
15      ind=find(tree(i).construction.alldist ...
16      ==min(tree(i).construction.alldist),1,'first');
17      tree(i).nodeinfo.sibling = neigh(ind); % make it as sibling
18      tree(i).nodeinfo.leaves = []; % no leaves
19      tree(i).nodeinfo.iteration = 0;
20  end
```

updateStruct

The ***updateStruct*** function implements the behaviour depicted in Figure 2.9. In this function, tree regions are merged according to the shortest distance between all siblings

for all tree nodes, this defines the *merging order*, which is an array of tree node labels to be merged together in ascending order. However, as discussed in Chapter 2, small and meaningless regions are prioritized to be merged first. Regions are considered small if their size is less than a percentage (alpha) of the mean region size for the initial segmentation map.
In Listing 3.2, the MATLAB function starts by sorting all tree node sibling distances while keeping label information. Next, an inspection for small region sizes is done and the merging order is updated accordingly.

Code listing 3.2: Update node structures - Step 1.

```
    D = [tree.construction];
    D = [D.dist]; % list of all distances
    [D,merging_order] = sort(D,'ascend');
 % Step 1: find the merging order
    alpha = 0.15;
    Nregions = length(merging_order);
    mean_size_region = size(map,1)/Nregions; % mean size of a region
    thresh = round(alpha*mean_size_region); % threshold for small regions
    F = regsize(merging_order)<thresh;
    % F is a flag binary vector where 1 means that the concerned regions gets
    % the merging priority
    merging_id = find(F,1,'first'); % find the ID of the first 1 in F
    if isempty(merging_id)
    % if no siblings have priority merging, merge the first two siblings
        merging_id = 1;
    end
```

The next step involves retrieving the two merging nodes information by making use of *merging_id* as in Listing 3.3. The following step creates a new node structure where the merging occurs. Lines 13-23 perform the sibling node merging for the ***descriptors*** field. The *updatenewnode* function in Line 24 updates the ***nodeInfo*** and ***construction*** fields of the tree. This is done by marking node i and node j as the children of the new label, transferring node i and node j neighbours to the new node, recalculating SAM distances between the new node and those neighbours, and finally, marking node i and node j as leaves if both nodes do not have leaves from a previous update. The merging order array is then updated and the next two siblings are fetched and merged. The loop keeps going until two regions are remaining.

Code listing 3.3: Update node structures - Steps 2, 3, & 4.

```
1  %Step 2: retrieve the two regions which are to merge
2  for i=1:N_iteration-2   % looping until there are two regions remaining
3      % make a copy of the merging nodes
4      node_i = tree(merging_order(merging_id));
5      node_j = tree(S(merging_id)); % S is a list of siblings
6      %empty the info about the first node merged
7      S(merging_id) = []; merging_order(merging_id) = [];
8      D(merging_id) = []; F(merging_id) = [];
9      % Step 3: Create the new structure in tree
10     newlabel = N_iteration+i;
11     tree(newlabel).label = newlabel;
12     % merge the two models by multiplying the two models by their sizes
13     Si = node_i.descriptors.size;
14     Sj = node_j.descriptors.size;
15     tree(newlabel).descriptors.size = Si+Sj;
16     %field model
17     Ri = node_i.descriptors.model; Rj = node_j.descriptors.model;
```

```matlab
    tree(newlabel).descriptors.model = (Si*Ri+Sj*Rj)/(Si+Sj);
    %field boundingbox
    bbi = node_i.descriptors.boundingbox;
    bbj = node_i.descriptors.boundingbox;
    D.boundingbox = [min(bbi(1),bbj(1)) max(bbi(2),bbj(2)) ...
        min(bbi(3),bbj(3)) max(bbi(4),bbj(4))];
    updatenewnode(node_i.label,node_j.label,newlabel);
    tree(newlabel).nodeinfo.iteration = i;
    % Step 4: update merging order
    % If priority function, update only when no regions with priority remain
    if all(F==false)
        [D,order] = sort(D,'ascend');
        merging_order = merging_order(order);
        S = S(order);
    end
end
```

The fusion of the last two regions is a simple merge procedure similar to Listing 3.3, however, in this case, it is required to identify the root of the tree as the last element on the tree structure by setting the node's ***height*** to 1.

completeStruct

This subfunction finalizes the buildBPT function by representing the tree structure as parents and children nodes. This is done by traversing the tree from the root children to the leaves (last children) such that the ***branch*** of each node is formed by a list of it's parents as depicted in Listing 3.4. First, the list of children is copied to *child* in Line 1. Then, the parent label of the two children is found and added to the branch of the children's labels. For example, the labels of the root of the tree and it's 2 children are 23833 and [23832,23811], respectively. Then, the parents list of Line 4 for both children will be just 23833 since the root does not have a branch (see the definition of ***branch*** in ***initStruct***). The child list is then updated with the children of 23832 and 23811.

Code listing 3.4: Final step in buildBPT.

```matlab
1  child = tree(end).nodeinfo.children;
2  while ~isempty(child);
3      parentlabel = tree(child(1)).nodeinfo.parent;
4      parents = cat(2,parentlabel,tree(parentlabel).nodeinfo.branch); % add to list
5      tree(child(1)).nodeinfo.branch = parents;
6      tree(child(1)).nodeinfo.height = length(parents)+1;
7      tree(child(2)).nodeinfo.branch = parents;
8      tree(child(2)).nodeinfo.height = length(parents)+1;
9      if ~isempty(tree(child(1)).nodeinfo.children); % if the 1st child have children
10         child = cat(2,child,tree(child(1)).nodeinfo.children);
11         % child list has the childinfo for the 1st child
12     end
13     if ~isempty(tree(child(2)).nodeinfo.children);
14         child = cat(2,child,tree(child(2)).nodeinfo.children);
15     end
16     child([1 2]) = []; %the child of the previous parent are deleted
17 end
```

The tree is then passed to pruning stage.

3.1.3 BPT Pruning

This function extracts a given number of regions (P_R) which will constitute a partition featuring the P_R most dissimilar regions created during the construction of the BPT. A new indicator field is added to the tree structure, called ***pruning***, such that 1 indicates that a node i is to be pruned, otherwise, not pruned. If P_R is greater than the number of leaves, then all the leaves will be pruned and the tree will not be used. In other words, the initial segmentation map is passed to the k-means directly. This is because the number of regions in the initial segmentation map is the number of leaves on the tree as established earlier in Chapter 2. Otherwise, the ***pruneBPTnbregions*** function listed in Listing 3.5 is called. The ***iteration*** assigned to each tree node keeps track of the build order which will be used to prune the tree. Lines 4-6 initializes the ***pruning*** field for all tree nodes to 0. Tree nodes are then pruned based on their build iteration. The number of required regions is interpreted as (root iteration - P_R) iterations, called iteration cut (*IterCut*). The function then finds the node such that it has a build iteration number greater than or equal to iteration cut. In other words, the node on a branch that is closest to the root in addition to the most recent built node.

Code listing 3.5: BPT Pruning.

```
1  function tree = pruneBPTnbregions(tree,NbReg)
2  N = length(tree);
3  Nbleaves = (N+1)/2;
4  for i=1:N
5      tree(i).pruning = 0;
6  end
7  IterMax = tree(end).nodeinfo.iteration;
8  IterCut = IterMax - NbReg;
9  i = N-1;
10 while (tree(i).nodeinfo.iteration >= IterCut)
11     tree(i).pruning = 1;
12     i = i - 1;
13 end
14 end
```

An important notion to understand is the difference between a segmentation map and a cluster map. In a segmentation map, the set of pixels which are not spatially adjacent but share the same cluster label will always be considered as different segments/regions. But in a cluster map, pixels with same cluster label can be spatially connected as well as can be located elsewhere in the image space. Hence, the regions on the segmentation map are not necessarily equal to the number of clusters, since 1 or more region can have the same cluster label (a roof label can be placed on different buildings on an image). In addition, if $P_R < k$, then the k-means clustering will not have enough regions to cluster. On the other hand, if $P_R \gg k$, then the k-means clustering will independently cluster the labels by merging regions that are similar, hence, the segmentation stage does not have an effect on the outcome of clustering.

The P_R parameter plays a great role in the classification accuracy. Different P_R regions generate different segmentation maps as shown in Figure 3.5 for the PaviaU dataset with 9 different classes, which essentially means that the number of clusters, k, is also 9 (datasets are discussed more in Chapter 4). With $P_R = 3$, a large number of regions are left out and hence the classification will not be able to label those regions. For $P_R = 10$,

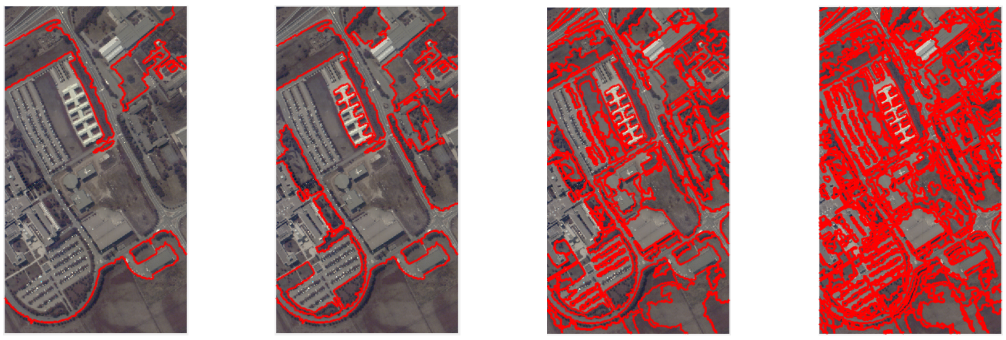

(a) $P_R = 3$. (b) $P_R = 10$. (c) $P_R = 100$. (d) $P_R = 600$.

Figure 3.5: Variation of final segmentation maps for PaviaU image.

the number of regions is close to the number of clusters but still, there might be more regions having the same label. This creates more possibility of classification errors. As for P_R values between 50 and 100, the segmentation map presents meaningful partitions of the image as it can be observed. Using $P_R = 600$ might be time consuming for the next stage to cluster those regions by giving a large number of regions the same label; the effect of BPT is unnoticeable. The effect of P_R on classification accuracy is further discussed in Chapter 4. By the end of this stage, a well refined segmentation map of the hyperspectral image is obtained as depicted in Figure 3.1.

3.1.4 K-means Clustering

In this stage, CLUS-BPT performs an efficient k-means clustering specifically designed for tree-based data structures, like BPT. Hyperspectral images have high spectral resolution, i.e., very narrow band gap of such images, hence, there is high redundancy in the data which are taken from the neighboring bands. Therefore, to reduce the dimension and redundancy, principal component analysis (PCA) is utilized for dimensionality reduction of the hyperspectral image, specifically, for feature extraction purposes. It aims to reduce the size of the image by choosing information only from the significant bands. The extracted features are embedded with the segmentation map of the BPT stage. Throughout this work, the number of chosen PCA components is 1 (explained in Appendix A), thus, a 2D-data out of the hyperspectral cube is formed. However, the number of components can be set by the user to more than 1. The PCA process is independent of the segmentation stages and can be done in parallel.
An example illustrating the embedding of the segmentation map with the reduced dimension data is shown in Figure 3.6. The segmentation map in the figure is a set of boundaries between data points of the values obtained by PCA. The purpose of clustering then is to give labels to those formed regions by making use of their data points. This framework implements the ***Filter*** clustering function described in Algorithm 2. Initially, 3 new data fields, ***candidateSet***, ***wgtCent***, and ***count***, are associated with all pruned nodes of the BPT tree. The ***candidateSet*** is initialized with random centers in the range of the available data. The distances between the points and centers in this function are calculated using the Euclidean distance. The set is then pruned using the ***Filter*** function to obtain the final cluster map.

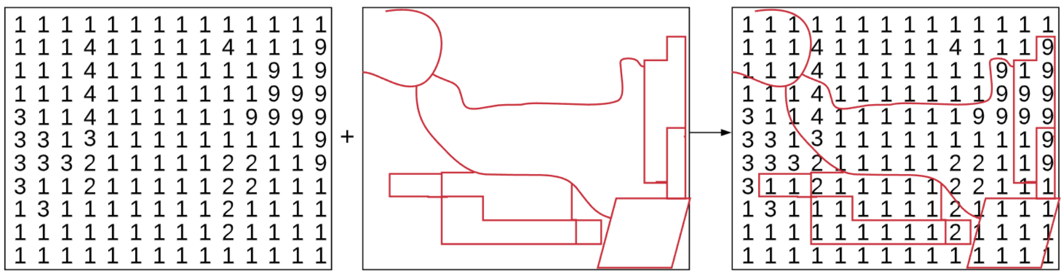


Figure 3.6: PCA and segmentation map embedding example.

3.2 Fast Spectral Clustering

Fast Spectral Clustering (FSC) method is based on incorporating spectral clustering with Nyström extension for efficient approximation of the eigenfunction problem. Algorithm 5 of section 2.6.5 describes the used FSC method. The framework is illustrated in Figure 3.7 and is implemented in this work based on the work of [37] in Python. Using only

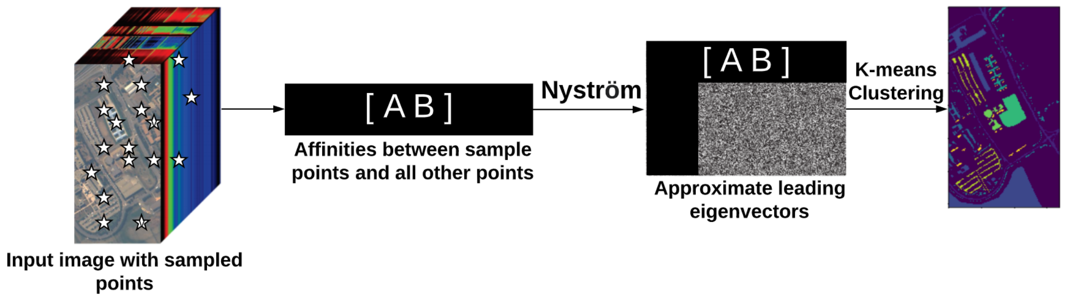

[A B]
Affinities between sample points and all other points
Nyström
[A B]
Approximate leading eigenvectors
K-means Clustering
Input image with sampled points

Figure 3.7: Flowchart of FSC.

those pixels marked by stars on the input image, a narrow strip of the full W matrix of Equation 2.21 is computed, shown as the $[AB]$ matrix. Each row contains the affinities from a sample point to the entire image. The Nyström extension allows one to then directly approximate the $\boldsymbol{C}$ matrix using the Laplacian matrix. Consequently, approximate the leading eigenvectors and cluster the image using k-means clustering as described by Algorithm 5.

Listing 3.6 implements FSC on input hyperspectral cube X. The loaded input has the dimensions of $pixels \times bands$. In addition, the function FSC takes $\boldsymbol{m}$ samples and the parameter ***sigma*** as inputs. It starts by randomly choosing samples from the input data of size m and accordingly create two subsets of data. Lines 11 and 12 make use of a built-in function to calculate the Euclidean distance between the points. ***d1*** and ***d2*** in Lines 19 and 21 correspond to the rows of $\hat{d}$ in Equation 2.32 of Chapter 2. Matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are updated using $\mathbf{d_i}, \mathbf{d_j}, \mathbf{d_{j+m}}$ in Lines 25 and 26. The Laplacian matrix $\boldsymbol{L}$ is then formed in Line 30. Singular component decomposition built-in function is used to obtain the eigenvectors of $\boldsymbol{L}$.

Line 34 constitutes the spectral embedding which is done by normalizing each embedding vector by first eigenvector ($V(:,1)$ selects all the rows of the 1st column of V). Algorithm code from [37] $embedding(:,i-1) = V(:,i)./V(:,1)$ is used. Finally, the first k eigenvectors are chosen which are then clustered using Python library built-in k-means clustering.

Code listing 3.6: Python code in LaTeX document

```python
def FSC(X, m, sigma, cosine=True)
    set_Z_tmp = X.ravel() % convert data to 1D array
    # randomly create the set Z based on the input data
    set_Z = np.random.permutation(set_Z_tmp)
    subset_X = set_Z[0:m]  # select the first m samples as A
    subset_Y = set_Z[m:]   # remaining part of the graph is B
    # selection of the subgraph
    Xm = X[subset_X]   # Xm size is m x m
    Xnm = X[subset_Y] # Xnm size is (n-m) x (n-m)
    # Euclidean distance in constructing the affinity matrix
    DXX = sc.cdist(Xm, Xm, 'sceuclidean', np.float64)
    DXY = sc.cdist(Xm, Xnm, 'sceuclidean', np.float64)
    Wxx = np.exp(-DXX/(2*sigma))     # affinity matrix A
    Wxy = np.exp(-DXY/(2*sigma))    #  affinity matrix B
    Wyx = Wxy.T                # affinity matrix B transpose
    # affinity matrix C = B^T A^-1 B
    # To extend to NCut, calculate the row sum of matrix W
    # Normalisation of Wxx and Wxy
    d1 = np.sum(np.vstack((Wxx, Wyx)), axis=0)    # A + B
    # B^T + B^T x A^-1 x B
    d2 = np.sum(Wyx, axis=0) + np.sum(Wyx, axis=0).dot(np.linalg.pinv(Wxx).dot(Wxy))
    dhat = np.sqrt(1 / np.hstack((d1, d2)))   # square root of d
    d_i = np.array([dhat[0:m]]) # di or dj
    d_j_m = np.array([dhat[m:]]) # d(j+m)
    Wxx = Wxx * (d_i.T.dot(d_i))   # updating A   Equation 2.19
    Wxy = Wxy * (d_i.T.dot(d_j_m)) # updating B   Equation 2.20
    Wyx = Wxy.T # B^T
    Wxx_inv = np.linalg.pinv(Wxx) # A^-1
    # Laplacian Matrix
    L = np.vstack((Wxx, Wxy)).dot(Wxx_inv).dot(np.hstack((Wxx, Wxy)))
    # Estimation of eigen vectors in L
    S, V, D = np.linalg.svd(L)
    k = 4 #number of clusters
    # Normalize each embedding vector by first eigenvector. Algorithm code was:
    # for i = 2:embedLength+1  # i starts from 2
    #    embedding(:,i-1) = V(:,i)./V(:,1);
    # end
    # The other option from the literature was to use this:
    # embedding_i,j = V_i+i,j./sqrt(Djj) -> Algorithm 5
    embed = numpy.zeros((V.shape[0], V.shape[1]))
    for i in range(k): # i starts from 0 # V[:,0] is 1st eigenvector
        embed[:,i] = numpy.divide(V[:,(i+1)], V[:,0])
    Xres = np.zeros((embed.shape[0], embed.shape[1]))
    idx = S.argsort()[::-1]
    eigenValues = S[idx] #sorted Eigenvalues
    embed = embed[:,idx]
    Xres[:, k] = embed  # choose the first k sorted embeddings
    km = KMeans(n_clusters=k)
    km.fit(Xres)
    y = km.predict(Xres)
    return y
```

Chapter 4

Software Results

In this chapter, clustering performance of the segment-based clustering using Binary Partition Trees framework (described in section 3.1) and the Fast Spectral Clustering (described in section 3.2) is tested and compared with one other state-of-the-art clustering method. The experiments are performed on eight different hyperspectral images. To evaluate their performance, metrics such as Normalized Mutual Information (NMI) and Purity are used. This analysis serves as a basis for the algorithm choice for FPGA implementation.

4.1 Software Results for CLUS-BPT and FSC

To assess the effectiveness of the CLUS-BPT and the FSC method, it is compared with the segment-based projected clustering using mutual nearest neighbour (PCMNN) method [18] discussed earlier in section 2.3.2. Note that for Fast Spectral Clustering (FSC), the values mentioned are the reported ones from the HSI spectral clustering implementation by [12] since it provided more accurate results than the implementation of Listing 3.6 and hence, more challenging results.
Further, to test the performance of the proposed frameworks, evaluation is carried on HSI datasets, and then several useful analysis are discussed. The experiments were ran under the same environment: Intel(R) Core(TM) i7-5930K CPU, 3.50 GHz, 64 GB memory, Windows 10 OS, Matlab version R2019a and Python 3.7.3.

4.2 Experimental Datasets

Experiments were conducted on eight widely used hyperspectral datasets.

- **Salinas** scene was collected by the 224-band AVIRIS (Airborne Visible/Infrared Imaging Spectrometer) sensor over Salinas Valley, California, and is characterized by high spatial resolution (3.7-meter pixels). Salinas covers 512 lines by 217 samples at as scale of 512×217. Salinas ground truth contains 16 classes.
- **Salinas-A** is a small subscene of Salinas image and it comprises 86×83 pixels located within the same scene as Salinas at [samples, lines] = [591–676, 158–240]. It includes vegetables, bare soils, and vineyard fields. Salinas-A ground truth contains 6 classes.

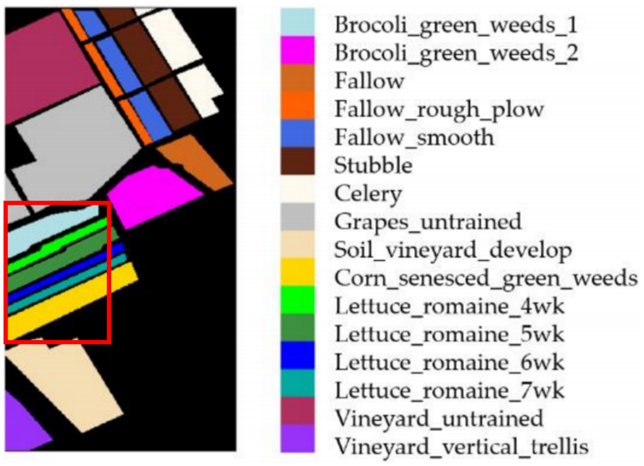


Figure 4.1: Salinas scene ground truth map and labels reference. Salinas-A is highlighted with a red box [47]

- **PaviaC** is acquired by the ROSIS (Reflective Optics System Imaging Spectrometer) sensor over the city center of Pavia (referred as PaviaC), central Italy. After removal of noisy bands, 102 bands were used for classification. The original dimension of the image, 1096 × 715 with the spatial resolution of 1.3 m, is used in this experiment. The dataset consists of nine land cover classes.

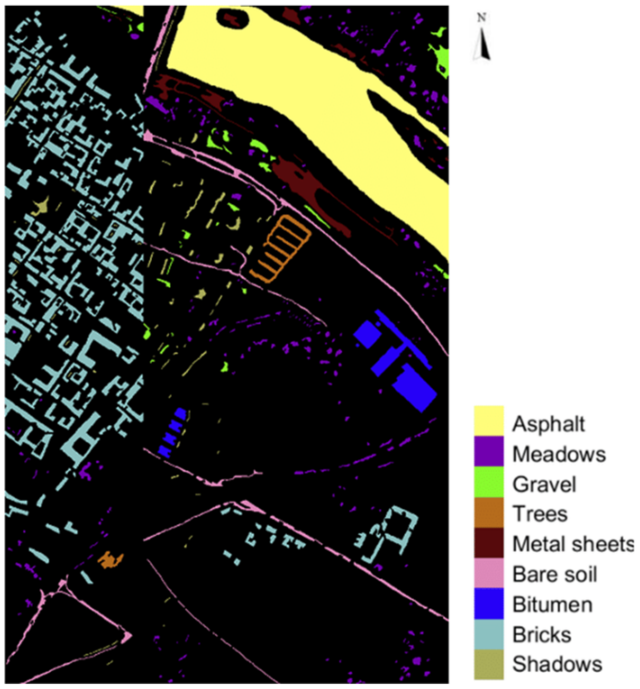


Figure 4.2: Pavia Center scene ground truth map with labels reference [47]

- **PaviaU** is also acquired by the ROSIS senser over University of Pavia, central Italy. The image of Pavia University is of a size of 610×340 and it has a spatial resolution of 1.3 m. A total of 115 spectral bands were collected, at the range 0.43–0.86 μm. Twelve spectral bands were removed due to noise and the remaining 103 bands were used for classification. The ground reference image available with the data has nine land cover classes.

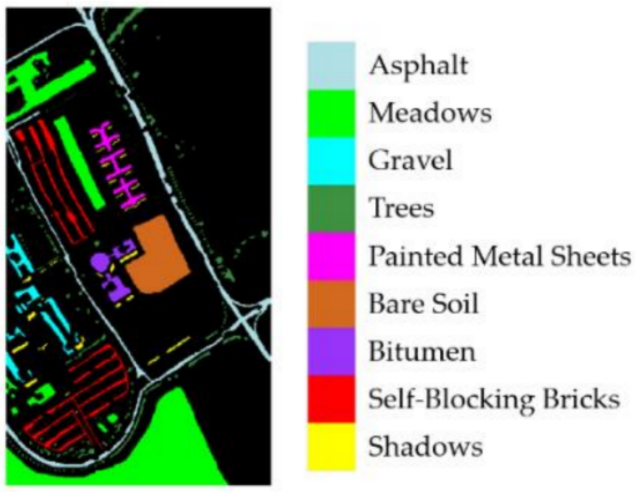


Figure 4.3: Pavia University scene ground truth map with labels reference [47]

- **Indian Pines** scene was gathered by AVIRIS sensor over the Indian Pines test site in North-western Indiana and consists of 145 × 145 pixels and 224 spectral reflectance bands in the wavelength range $0.4 - 2.5\ 10^{-6}$ meters. The scene contains two-thirds agriculture, and one-third forest or other natural perennial vegetation. There are two major dual lane highways, a rail line, as well as some low density housing, other built structures, and smaller roads. The ground truth available is divided into sixteen classes.

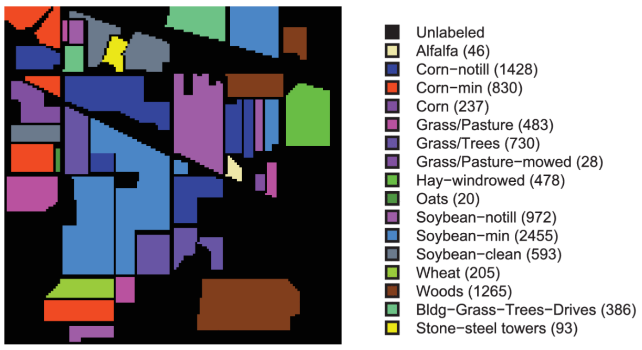


Figure 4.4: Indian Pines ground truth map and labels reference [47]

- **Samson** scene is an image with 95 × 95 pixels. Each was recorded at 156 channels covering the wavelengths from 401 nm to 889 nm. The spectral resolution is high up to 3.13 nm. There are three target end-members in the dataset, including “Rock”, “Tree”, and “Water”.

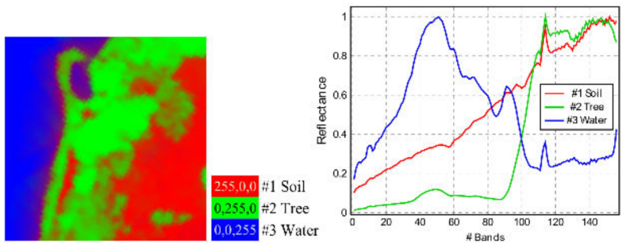


Figure 4.5: Samson scene ground truth map and labels reference [47]

- **Jasper Ridge** is one of the most widely used hyperspectral image datasets, with each image of size 100 × 100 pixels. Each pixel was recorded at 198 effective channels with the wavelengths ranging from 380 to 2500 nm. The spectral resolution is up to 9.46 nm. There are four end-members latent in this dataset, including “Road”, “Soil”, “Water”, and “Tree”.

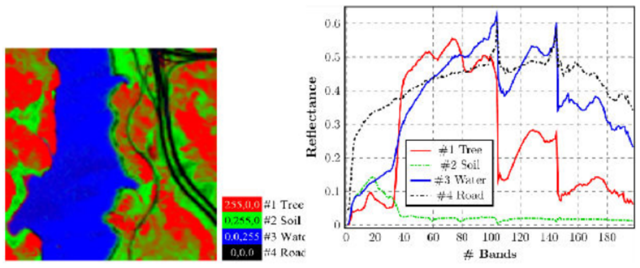


Figure 4.6: Jasper Ridge ground truth map and labels reference [47]

- **Urban** scene consists of images of 307 × 307 pixels with spatial resoultion of 10 m. Each pixel was recorded at 210 channels with wavelengths ranging from 400 nm to 2500 nm. There are three versions of the ground truth, which contain 4, 5 and 6 end-members respectively. In this experiment, we use 4 end-members including "Asphalt", "Grass", "Tree", and "Roof".

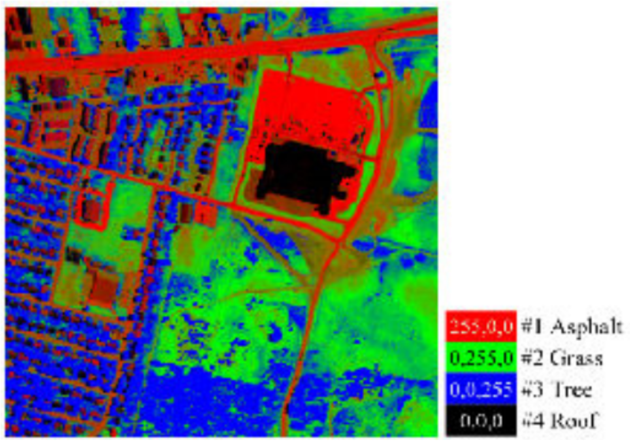


Figure 4.7: Urban dataset scene ground truth map and labels reference [47]

4.3 Evaluation Metrics

In the experiments, clustering results are evaluated by Purity and Normalized Mutual Information (NMI). Let C be the set of classes obtained from ground reference information and ω be the set of clusters obtained from a clustering method/framework.

- Purity is an external evaluation criterion of cluster quality. It is the most common metric for clustering results evaluation and can be defined as

$$Purity(\omega, C) = \frac{1}{n} \sum_i \max_j |\omega_i \cap C_j| \tag{4.1}$$

  where n denotes the number of data points. The worst clustering result is very close to 0 and the best clustering result has a purity of 1. Purity counts the number of correctly assigned points for each cluster and divides the total by n where n is the total number of points.
- NMI is a normalization of the mutual information score (MI). MI can be obtained as:

$$MI(\omega, C) = \sum_i \sum_j p(\omega_i \cap C_j) \log_2 \frac{p(\omega_i \cap C_j)}{p(\omega_i).p(C_j)} \tag{4.2}$$

  where $P(\omega_k)$, $P(C_j)$, and $P(\omega_k \cap C_j)$ are the probabilities of a point being in cluster ω_k, class C_j, and in the intersection of ω_k and C_j, respectively. Then the NMI can be obtained as follows:

$$NMI(\omega, C) = \frac{MI(\omega, C)}{\max[H(\omega), H(C)]} \tag{4.3}$$

  where $H(\omega) = -\sum_i p(\omega_i) \log_2 p(\omega_i)$ and $H(C) = -\sum_j p(C_j) \log_2 p(C_j)$ are the entropies of ω and C, respectively. The larger is the NMI, the better is the clustering result.

High purity is easy to achieve when the number of clusters is large - in particular, purity is 1 if each point gets its own cluster. Thus, we cannot use purity to trade off the quality of the clustering against the number of clusters. NMI allows us to make this trade off since it is normalized. The main advantage of using NMI evaluation metric is that since it is normalized, it can measure and compare the accuracy between different cluster maps having different number of clusters. Thus, an unsupervised clustering method does not need to know the number of clusters of data beforehand for evaluation. For example, it is possible to calculate the mutual information (accuracy/correlation) between a cluster map with 6 clusters and another cluster map with 4 clusters.

4.3.1 Parameter Settings

CLUS-BPT requires input for two parameters P_R and k as shown earlier in Table 3.1 . First, for all the experimented HSI images, NMI values are obtained for the proposed method by varying the number of pruned regions P_R. P_R is varied from k to $5 \times k$ regions, in steps of k for all HSI images except PaviaC, PaviaU, and Urban. In the case of PaviaC and PaviaU, P_R is varied from 150 to 400, in steps of 50 regions. For Urban, P_R

is varied from 4 to 128, in steps of P_{R_i} where i is the current step number.
Identification of k for a data set is a non-trivial task. Hence in this study, for all the hyperspectral images, accuracies are obtained for the proposed method CLUS-BPT by varying the value of k in steps of 2. In case of Salinas-A, k is varied from 6 to 16, for PaviaU and PaviaC, k is varied from 9 to 19. k is varied from 16 to 26 for both Indian Pines and Salinas. For Samson image, k is varied from 3 to 13 and for Urban and Jasper Ridge images, k is varied from 4 to 14. Throughout the experiements for CLUS-BPT, the number of principal components (PCA) are chosen to be 1 such that they are able to account for at least 99% of the total variance in the image.
Variation of k further proves that the framework is unsupervised. Otherwise, the framework will be instructed to cluster for a specific k number of clusters.

4.4 Results and Comparisons

Initially, the influence of the number of regions on the NMI values obtained for the proposed method is investigated. Figure 4.8 illustrates the variation of NMI with respect to the different number of regions (P_R) of the segmentation map. These values of NMI are obtained by setting k equal to the number of classes in an image and the values represent the average of 10 runs for each different setting. From Figure 4.8, it can be observed that different images have different values for P_R at which the NMI is the highest. Furthermore, Salinas-A image had the highest NMI at $P_R = k = 6$. A downward trend is noticeable for PaviaC, Salinas-A, and Indian Pines images as number of regions increases. However, in case of PaviaU image, from Figure 4.8 (a), it can be observed that the quality of clustering increases with more segments across the segmentation map, whereas for Jasper Ridge and Urban images, at $k = 4$, the NMI values tend to converge around a certain value. Once the optimal value of P_R is determined for every image, NMI values are obtained at different number of clusters for the proposed CLUS-BPT method.

4.4.1 Effect of Number of Clusters

Figures 4.9 and 4.10 show the behavior of NMI as a function of increasing k for all the experimented images. From Figure 4.9, it can be observed that the proposed method scored the highest NMI when k is the original number of clusters for almost all the images. However, for PaviaC, the peak NMI value of 0.5806 is achieved at $k = 17$ for the proposed method. In the case of Salinas and Indian Pines images, CLUS-BPT scored the peak NMI values at $k = 26$ and $k = 20$ respectively.

Table 4.1 reports the best NMI values and corresponding purity values for all the compared methods on Salinas and PaviaU images. For Salinas, the values shown for the proposed method, are the best values corresponding to k which have achieved the highest NMI value. Both PCMNN and the proposed method achieved the highest NMI at $k = 26$. The Salinas image cluster map obtained by the proposed method is shown in Figure 4.11(d). Figure 4.11(d) reveals that the proposed method identified all the 16 classes of Salinas image except grapes untrained, which is assigned to vineyard untrained class.

In the case of PaviaU, Table 4.1 shows that all methods had close accuracies. CLUS-BPT scored the highest NMI value of 0.5806 at $k = 17$ while FSC scored the lowest,

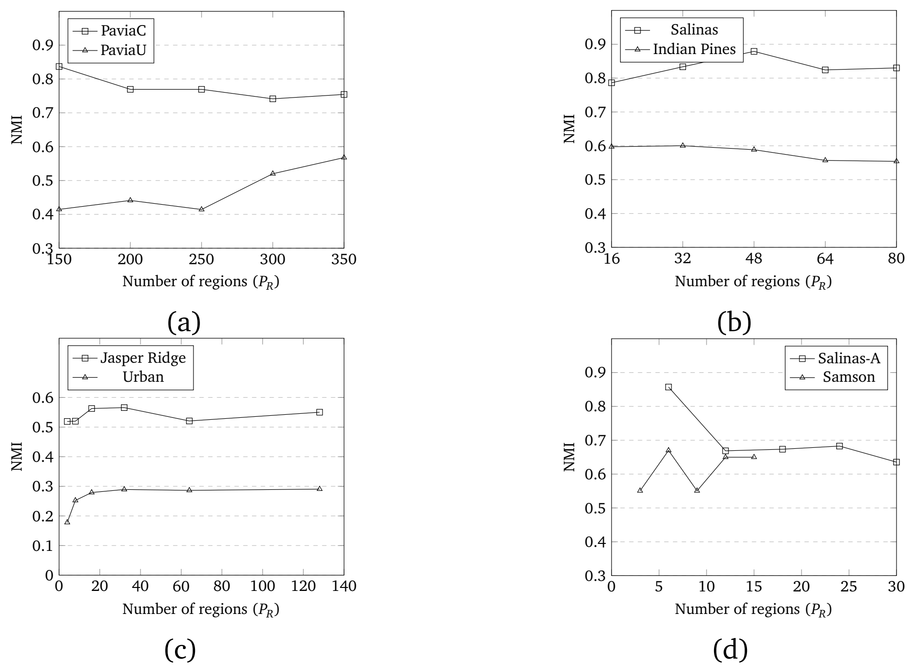


Figure 4.8: Average NMI values (10 runs) versus the number of regions (P_R) for (a) PaviaC and PaviaU, (b) Salinas and Indian Pines, (c) Jasper Ridge and Urban, and (d) Salinas-A and Samson.

0.5654 at $k = 13$. The cluster maps corresponding to the optimum k mentioned in Table 4.1 for the proposed method are shown in Figure 4.11(h).

For Indian Pines and Salinas-A, Table 4.2 shows that CLUS-BPT resulted in higher NMI and purity values compared to FSC method. CLUS-BPT scored the highest NMI value at $k = 20$ and $k = 6$ for Indian Pines and Salinas-A, respectively. The cluster maps corresponding to the optimum k for the proposed method are illustrated in Figures 4.11(b) and 4.11(f).

Table 4.1: Best purity and NMI values obtained by PCMNN, FSC, and CLUS-BPT for Salinas and PaviaU images.

Dataset / Framework	Salinas			PaviaU		
	Purity	NMI	k	Purity	NMI	k
PCMNN [18]	-	0.8586	26	-	0.5654	13
FSC [12]	0.62	0.72	16	0.61	0.57	9
CLUS-BPT^{a,b}	0.7638	0.8882	26	0.6996	0.5806	17

^{a}Values are obtained at $P_R = 48$ for Salinas.
^{b}Values are obtained at $P_R = 400$ for PaviaU.

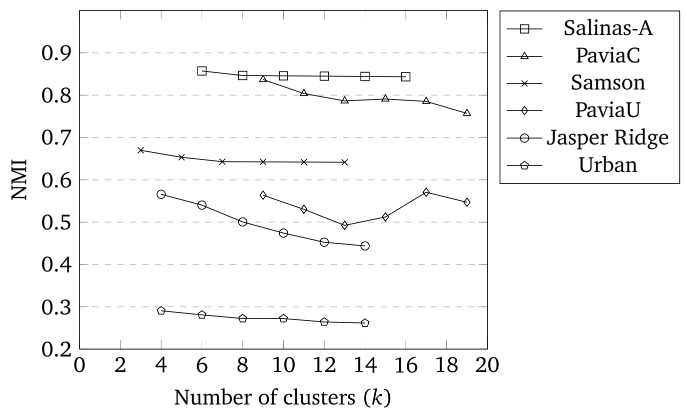


Figure 4.9: NMI values obtained at different number of clusters for the proposed method.

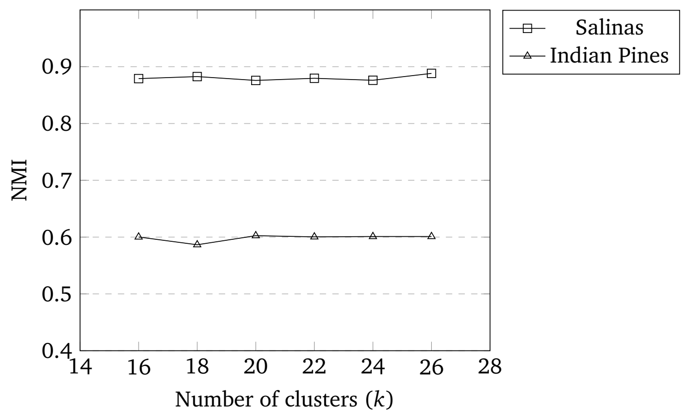
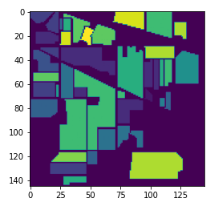
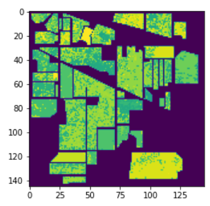


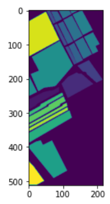
Figure 4.10: NMI values obtained at different number of clusters for the proposed method, for Salinas and Indian Pines images.

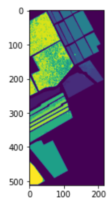
Table 4.2: Best purity and NMI values obtained by FSC and CLUS-BPT for the rest of the images.

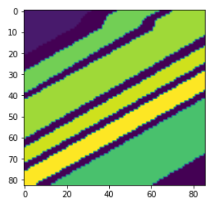
Framework / Dataset	FSC [12]			CLUS-BPT			
	Purity	NMI	k	Purity	NMI	k	P_R
Indian Pines	0.46	0.49	16	0.5758	0.6025	20	32
Salinas-A	0.80	0.81	6	0.8753	0.8572	6	6
Samson	0.85	0.75	3	0.6896	0.6698	3	6
Urban	0.51	0.21	4	0.90	0.2906	4	128
Jasper Ridge	0.83	0.71	4	0.7652	0.5658	4	32
PaviaC	-	-	-	0.8412	0.8369	9	150

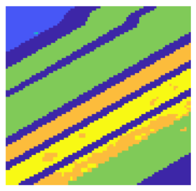

(a) GT (Indian Pines)

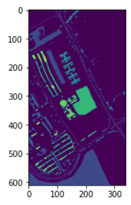

(b) Result (Indian Pines)

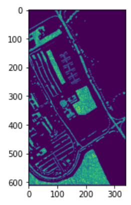

(c) GT (Salinas)

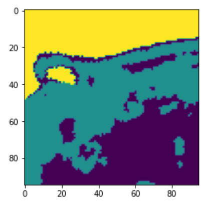

(d) Result (Salinas)

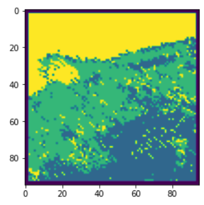

(e) GT (Salinas-A)

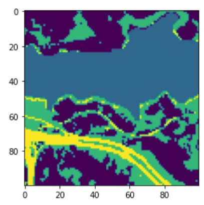

(f) Result (Salinas-A)


(g) GT (PaviaU)


(h) Result (PaviaU)


(i) GT (Samson)


(j) Result (Samson)


(k) GT (Jasper Ridge)

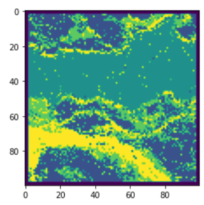
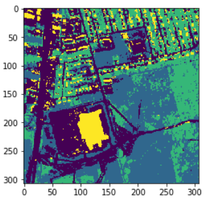

(l) Result (Jasper Ridge)

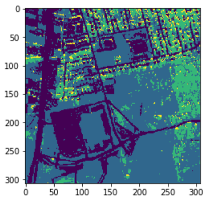
Figure 4.11: HSI ground truth and results.

Fast spectral clustering resulted in less number of misclassification errors for Samson and Jasper Ridge. However, CLUS-BPT scored 90% purity for Urban image. This

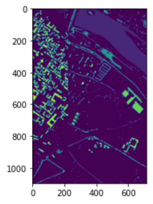
means that each cluster C_i in CLUS-BPT result has identified a group of pixels as the same class that the ground truth has indicated. This can be seen on Figures 4.12 (a) and (b) where the only misclassified class out of the total 4 is the roof (marked in yellow) which makes a small part of the image.
Cluster label coloring is different between PaviaC ground truth and result as it can be observed in Figures 4.12 (c) and (d), while the accuracy is high as indicated by purity and NMI in Table 4.2. This is because clustering algorithms do not necessarily learn the specific label number, hence, class label numbers of ground truth and cluster results may not match but may point to the same object on an image.



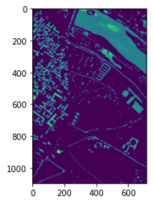
(a) GT (Urban)



(b) Result (Urban)



(c) GT (PaviaC)



(d) Result (PaviaC)

Figure 4.12: HSI ground truth and results.

4.4.2 Computational Time

Figure 4.13 lists the computational time on Salinas, PaviaU, Salinas-A, and Indian Pines datasets as experimented on the proposed CLUS-BPT method. The computational time is calculated as a function of increasing number of regions P_R from 50 to 1000, in steps of 50. Further, the best achieved number of clusters k is fixed for every image which corresponds to the k as indicated in Tables 4.1 and 4.2 for Salinas, PaviaU, Salinas-A, and Indian Pines, respectively. We ran the experiments under the same environment: Intel(R) Core(TM) i7-5930K CPU, 3.50 GHz, 64 GB memory, Windows 10 OS, and Matlab version R2019a.

From Figure 4.13, it can be observed that the computational time grew rapidly along with the increase of the number of regions for Salinas and PaviaU images while the proposed method performed better for Indian Pines and Salinas-A images. This difference owes to having images with high spatial dimensions, such as 512×217 and 610×610 for Salinas and PaviaU, respectively. On the other hand, Salinas-A has an image dimension of 86×83 which is about $7\times$ smaller than Salinas and PaviaU. Figure 4.13 shows a near linear execution time for Salinas image. It can be concluded that large spatial dimension images lead to large number of regions to be processed, hence, increases the computational time. A slight improvement can be made by parallelizing the PCA stage since it depends on the input data alone.

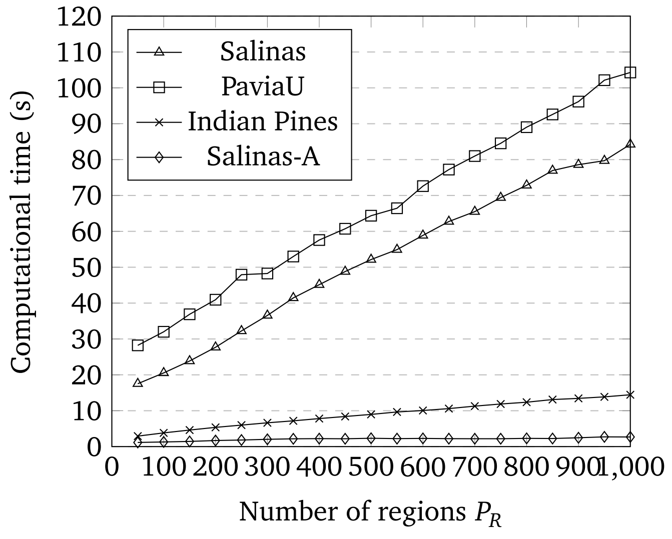


Figure 4.13: Computational time measured at different number of regions P_R for the proposed method.

Table 4.3: Computational time for FSC and CLUS-BPT obtained for the best purity and NMI results for all datasets.

Framework / Dataset	FSC CT(s) [12]	CLUS-BPT CT(s)
PaviaU	1.34	58.2
Salinas	1.62	7.48
Indian Pines	0.53	2.72
Salinas-A	0.17	1.56
Samson	0.10	1.52
Urban	3.01	2.15
Jasper Ridge	0.11	2.78
PaviaC	-	18.26

Table 4.3 illustrates the average time taken in seconds to complete the execution experiments on eight publicly available datasets as reported by FSC [12] and experimented on CLUS-BPT. The displayed values are the average of ten runs. These values are obtained by setting the parameters as indicated in Tables 4.1 and 4.2. As it can be observed, CLUS-BPT framework took more time than FSC to output the best results in almost all the datasets except for Urban and PaviaC images. PaviaU image experiment achieved the largest timing difference while the difference in clustering accuracies between CLUS-BPT and FSC are not substantial. On the other hand, the clustering accuracy difference is significant for Salinas image and hence the large timing difference can be justified. The execution time difference between CLUS-BPT and FSC is smaller for the other datasets including Indian Pines, Salinas-A, Samson, and Jasper.

4.5 Algorithm Choice

In order to choose the adequate algorithm for FPGA implementation, experiments of both methods were carried out in this chapter on eight different HSI datasets. Evaluation performance included purity, NMI, and computational time. According to the following observations made on Tables 4.1, 4.2, and 4.3, the Fast Spectral Clustering (FSC) method is chosen for FPGA Implementation:

- Fast Spectral Clustering (FSC) execution time on almost all datasets is much lower than that of CLUS-BPT. On some datasets, the computational time is near real-time.
- Classification performance of both methods are quite comparable with slight positive difference for CLUS-BPT. Hence, the choice is a trade-off between speed and accuracy.
- CLUS-BPT involves a long list of components including watershed segmentation, BPT construction, BPT pruning, PCA, and filtering algorithm for k-means clustering. While on the other hand, FSC uses less number of components and also has the advantage of deploying numerical approximations.
- In addition, FSC involves matrix processing functions which has been implemented on FPGAs in literature.

Chapter 5

HW/SW Co-design Implementation

5.1 Overall System Structure (HW/SW)

A HW/SW co-design solution for Fast Spectral Clustering of Algorithm 5 is developed for Zynq-7000 SoC. The block diagram of the implemented system is illustrated in Figure 5.1. The design is partitioned into PS and PL. The processing logic blocks are described in detail in the following sections. The design makes use of two BRAM modules to store the hyperspectral image cube and the constructed graph matrix *A*. In addition, the co-design also utilizes two different AXI DMAs to transfer vast amounts of data to and from two different hardware accelerators using appropriate AXI interfaces, including AXI4-Stream and AXI4-Lite.

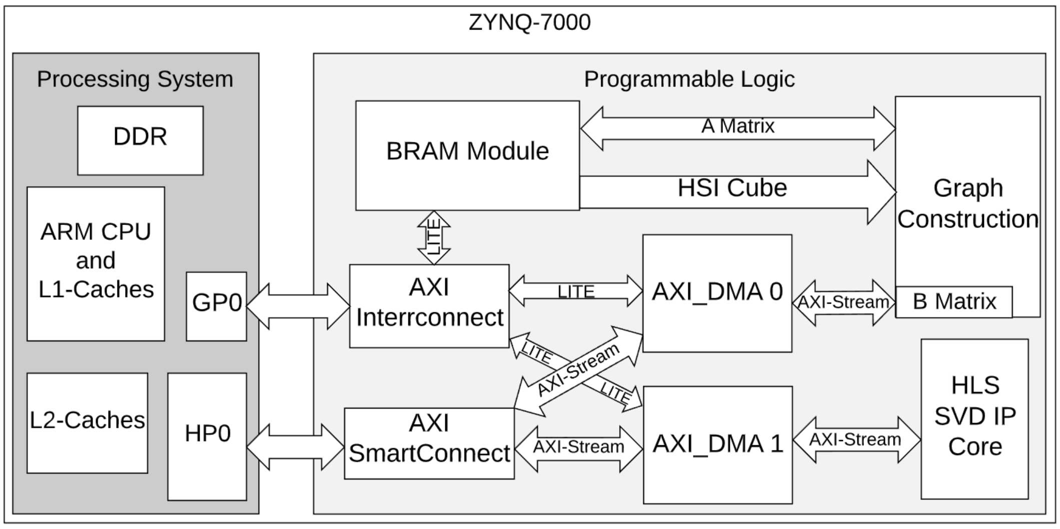


Figure 5.1: PS and PL Partitions in the Zynq-7000 HW/SW co-design.

The co-design solution goes through the following procedure where the PS is implemented in C programming language and the PL is implemented in VHDL:

1. Processing system is initialized.

2. Hyperspectral cube is loaded from the SD card to the DDR memory.

3. BRAM module and 2 AXI_DMAs transfers are configured via AXI-lite register file through AXI_Interconnect.

4. HSI cube is written to BRAM by making use of a General Purpose AXI master port (GP0) in the Zynq PS.

5. Graph Construction module reads the HSI data band per pixel, processes the affinity matrices A and writes it back to the BRAM. Meanwhile, matrix B is being computed and streamed through AXI_DMA_0.

6. Matrices A and B are read by the PS and saved in DDR memory.

7. The PS forms the Laplacian matrix as means of this expression $\begin{bmatrix} A \\ B \end{bmatrix} A^{-1} \begin{bmatrix} A & B \end{bmatrix}$ by making use of the necessary matrix multiplication and matrix inverse.

8. The High Performance (HP0) port of Zynq PS connected to the external DDR memory and is used along with AXI_DMA_1 to stream the Laplacian matrix to the high-level synthesis module of SVD.

9. The eigenvalue and eigenvector decompositions computed by SVD core are streamed back to the PS.

10. The PS C code sorts the eigenvalues and the corresponding eigenvectors. In addition, it clusters the sorted eigenvectors using K-means clustering.

11. Finally, clustering results are written to the SD card. Results can be plotted using MATLAB/Python.

Before loading HSI cube from the SD card to the DDR memory, the design requires data preprocessing, which is done by reducing the HSI spectral bands dimension to 16 using PCA on MATLAB. Generic data parameters are kept consistent throughout the project and are designed to be modular such that the generic parameters control the design size. Table 5.1 shows the generic parameters, their description, and example values used throughout the HW/SW co-design modules. The default values for pixel dimensions are based on the reduced Salinas-A dataset.

Table 5.1: Generic parameters and example values for HW/SW codesign.

Generic Parameter	Description	Default Value
NUM_BANDS	Number of bands in HSI	16
NUM_PIXELS	Number of pixels in HSI (Image dimension $x - by - y$)	4096
NUM_VALUES	NUM_BANDS × NUM_PIXELS	65536
M_SAMPLES	Number of random samples m	100
σ	sigma parameter in similarity matrix W	0.1
k	Number of clusters k	16
BRAM_DATA_WIDTH	Bit width of input and processing data	32
BRAM_ADDR_WIDTH	Address bit width of input HSI data	16
BRAM_ADDR_WIDTH_A	Address bit width of Matrix A data	14

5.2 BRAM for Input and Processing Logic

The BRAM module designed in this work is inspired by [48]. HSI data is transferred to and stored on the BRAM as shown in Figure 5.1. The BRAM module consists of block RAM modules and an AXI-Lite register file which is used to initialize and update the data stored in BRAM. Figure 5.2 shows two registers pointing to a specific BRAM module and one additional register (register 3) used as an adddress controller by the Zynq PS to read the Matrix A data. Register 1 and 2 point to two BRAM modules explained in section 2.7.4. Both modules are of the three port mode (2 reads and 1 write) type. One port is used for writing from PS/PL, and two read ports from PL.

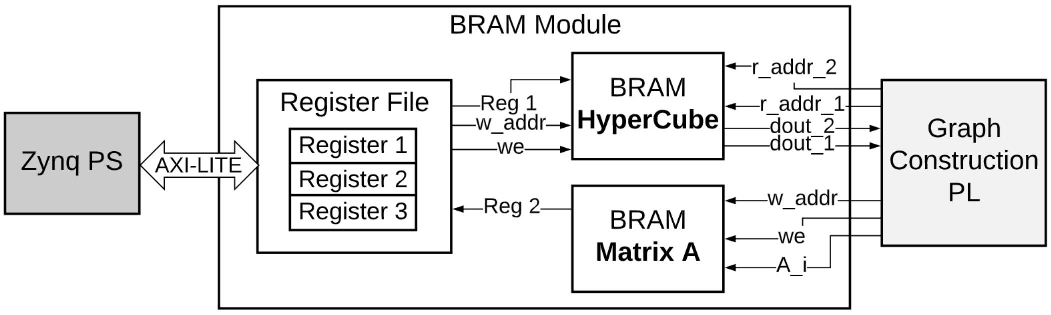


Figure 5.2: Block diagram of BRAM module with register file.

For the input logic, the BRAM module gets HSI data from the DDR (PS) by making use of a General Purpose AXI master port (GP0) which is connected to one AXI slave register on the BRAM module. This allows sequentially data send transfers from the Zynq PS.

As for the processing logic, the ***HyperCube*** input BRAM module comprises 2 read ports corresponding to 2 different read addresses initiated from the Graph Construction module discussed in the next section. The 2 read addresses output 2 HSI data pixels of size BRAM_DATA_WIDTH. It takes 1 clock cycle for each read transaction processed by the HSI BRAM module. In addition, the affinity matrix A is stored in the BRAM module using the corresponding write signals while graph matrix A formation is being processed. The register file used to handle the BRAM logic is shown in Table 5.2.

Table 5.2: AXI-lite register file description.

Register	Value	Description
1	Data	Writing HSI data.
2	Data	Reading matrix A.
3	Address	Address for reading matrix A row.

5.3 Similarity Graph Construction

5.3.1 Sampling

The first stage of spectral clustering involves sampling random pixels from the stored HSI data. This is done by producing a random number in the range of the BRAM matrix size holding the HSI data. The BRAM matrix size in this case is the BRAM address width ($log_2(Num\_Pixels \times Num\_bands)$). One popular way of generating pseudo-random numbers in HW is by means of an LFSR (Linear-Feedback Shift Register).

The input bit to the shift register is a linear function of its previous value as shown in Figure 5.3. The linear function consists of XOR of some bits of the overall shift register value. The study to generate these sequences is based on linear algebra, where the register is interpreted as a polynomial. Each bit in the register is the coefficient of order n of such a polynomial. Hence, a register of length n can generate a pseudo-random sequence of maximum length $2^n - 1$. In this case, the length of the LFSR is the BRAM address width used to store the HSI cube (BRAM_ADDR_WIDTH). Figure 5.3 shows an implemented LFSR corresponding to the Salinas-A data. The feedback linear polynomial function represented here is $x^{16} + x^{14} + x^{13} + x^{11} + 1$.

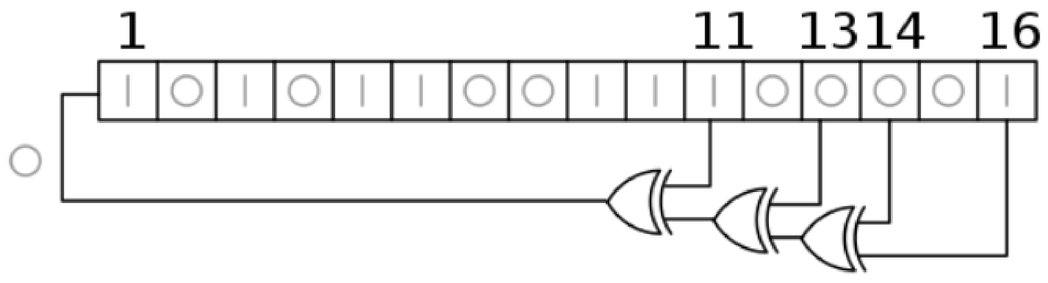


Figure 5.3: 16-bit LFSR.

The address produced by the LFSR is read by the BRAM module and the corresponding HSI data is fetched for the next stage. Sampling stage justifies the reason behind storing the HSI data on the BRAM module rather than streaming the data from the PS to the PL. Note that it takes 1 clock cycle to produce a new random address.

5.3.2 Graph Construction

In this solution, the similarity graph implemented of chosen m pixels of a hyperspectral image considers only the reflectance value of the pixels (x_i, x_j) for simplicity. The similarity between samples can be defined as:

$$A_{ij} = \exp \frac{-||x_i - x_j||_2^2}{2\alpha\bar{\sigma_x}^2}, \qquad i, j = 1, 2, \ldots, m, \tag{5.1}$$

where $\bar{\sigma_x}$ is defined as

$$\bar{\sigma_x} = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}||x_i - x_j||_2^2. \tag{5.2}$$

and α is a parameter to control the number of neighbors for each node in the affinity matrix.

The Euclidean distance : $||x_i - x_j||_2^2$ is usually calculated for a set of points (a vector) but in our case, it is between two points of 1 dimension and is simply $= \sqrt{(x_i - x_j)^2}$. This can be further simplified by the square root to be: $x_i - x_j$. Hence, calculating $||x_i - x_j||_2^2$ requires one subtraction unit.
A_{ij} can be further simplified by setting the parameters α and σ beforehand, to 10 and 0.1 respectively [12]. This leads to ($\exp \frac{x_j - x_i}{2}$). Division by 2 can be simply implemented by right shifting before applying the exponential function.

In literature, there are many methods to implement the normal use of the exponential function in hardware. A simple solution is to make use of a lookup table (LUT). A lookup table is basically a table that determines what the output is for any given input.
In this work, the exponential function is implemented using HDL Coder from MathWorks [49]. HDL Coder tool is very similar to Vivado HLS discussed in section 2.8, except that HDL Coder makes use of MATLAB code based functions instead. Given a particular range for x, the $exp(x)$ function is replaced with a lookup table. The range used in this design is (x : −10 to 10) and thus, the input reflectance values are scaled down accordingly during pre-processing of the HSI cube. The RTL design of the graph construction for either A or B matrix is depicted in Figure 5.4. As it can be observed, X_i is saved in a 32-bit register while X_j runs from 1 to m. The 32-bit register is updated when A_i is computed. This process is controlled by a finite state machine to control as shown in Figure 5.5.

The graph construction for matrix A is implemented using the finite state machine involves the following states:

- **Sampling**: This is where the LFSR is used to generate random addresses. The addresses are saved in a block RAM of size m samples.

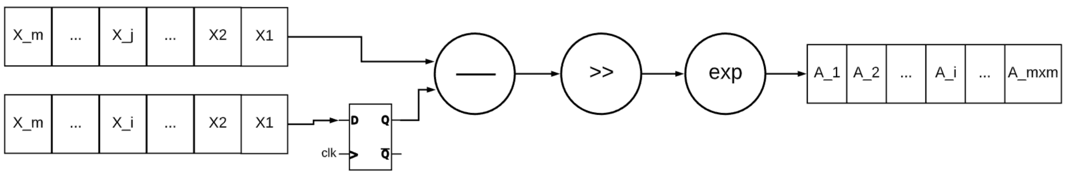


Figure 5.4: Implemented RTL design for graph A calculation.

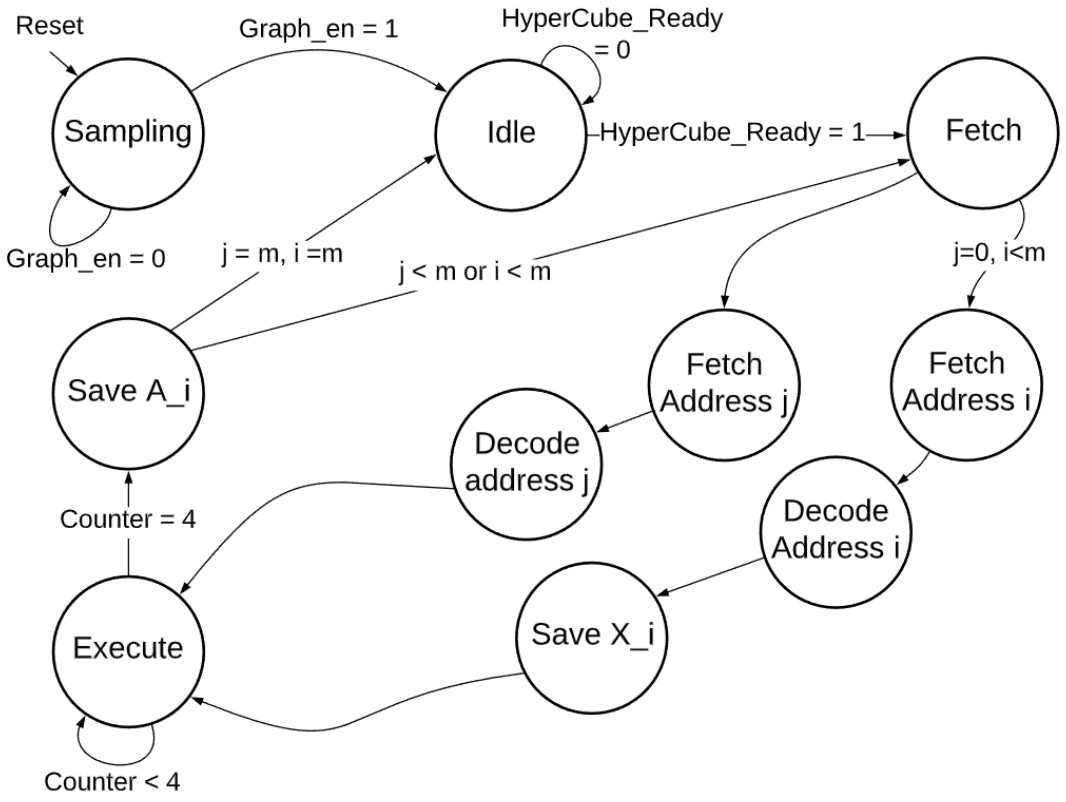


Figure 5.5: Implemented state machine for matrix A calculation.

- **Idle**: In this state, the state machine waits for the sampling process to be done and for the HyperCube data to be saved in the BRAM. This state is also reached when matrix *A* is fully calculated.
- **Fetch**: This state decides which 2 addresses *i* and *j* to get from the addresses BRAM initiliazed in the *Sampling* stage. Reading an address requires one clock cycle.
- **Decode**: The 2 chosen addresses are sent to the main BRAM module to select from the HyperCube data. It takes one cycle to get the X_i or X_j reflectance values.
- **Execute**: This is where A_i is calculated. It takes 4 clock cycles for subtraction, right shift, and 1 exponential function.
- **Save** A_i: In this state, A_i write address and actual value are sent to matrix A BRAM to save it. This state also decides whether to continue to the next i, j or the process of calculating matrix A is done.

As explained earlier, the state machine starts at *Sampling* on reset. Given the number of *m* samples, the state generates the random addresses and saves them in a BRAM. Concurrently, the BRAM module accepts the input HyberCube data sent from the Zynq PS, since the *Sampling* state does not depend on the input data rather, the size of the data which is set as a generic parameter (NUM_VALUES). *Graph_en* signals the *Idle* state that the random number generation is done and saved. Once the input data is loaded on the HSI BRAM, *HyperCube_Ready* signals the *Fetch* state that the full input data is captured and ready for processing.

Fetch state is directly reading from the BRAM of sample addresses. Counters i and j are pointers corresponding to the addresses of the BRAM sample addresses. However, read access is done per 1 instruction for either i or j and not concurrently. The idea of the coming states is to fix an address i and increment address j until m and then increment i for every $j = m$ reached until $i = m$.

If it is a new round/loop, then $j = 0$ and a new address i is fetched. The fetched address is sent to the BRAM to decode, which takes 1 clock cycle equivalent to the *Decode* state. Meanwhile, address j is fetched from the sample addresses BRAM. Next, the state machine goes to another *Decode* state to decode the j address. Meanwhile, the decoded X_i value is saved in a 32-bit register for usage in the coming loop. If it is not a new round/loop, then $j \neq 0$ and a similar *Fetch*, *Decode* process applies for address j.

The *Execute* state already has the saved X_i as one of the operands of the subtraction unit. After calculation of A_i, the value is written to the matrix A BRAM, which can be also read by the Zynq PS. The *Save* A_i state goes to *Fetch* if the calculation is not done ($i \neq m$), otherwise, it goes to the *Idle* state.

The critical path for the matrix A calculation is $9m^2 + 1$ with 1 subtraction unit, 1 LUT, and 1 right shift operation. m sampling clock cycles take place while HyperCube data input is being written to the BRAM so it is not considered as part of the critical path. The $+1$ represents the first Idle state. *Fetch* and *Decode* states take 1 clock cycle each and 1 clock cycle for saving X_i in a register. Furthermore, the critical path for 1 loop is 9 clock cycles with the longest path: Fetch $\rightarrow$ Fetch Address i $\rightarrow$ Decode Address i $\rightarrow$ Save X_i $\rightarrow$ Execute $\rightarrow$ Save A_i.

Matrix B calculation follows a similar RTL implementation as in Figure 5.4 as well as the finite state machine in Figure 5.5. However, for the RTL implementation, $i : 0 \rightarrow m$ and $j : 0 \rightarrow (n - m)$. Hence, resource-wise, matrix B computation requires the same set of resources as matrix A.
As for the FSM, matrix B calculation differs in the ***Save*** B_i stage. The size of B is $m \times (n - m)$ where $n \gg m = 128 \times (65536 - 128) = 8372224$ values, using the generic parameters. Hence, it is expensive, resource-wise, to store matrix B on the BRAM. Instead, it is saved in the DDR memory on Zynq PS. Master Output module, adapted from [48], streams the calculated matrix $\mathbf{B_i}$ to Zynq PS by communicating with AXI_DMA_0. The output module acts as an AXI stream master interface setting the different AXI4-Stream channels TDATA and TLAST. Moreover, the AXI stream on the DMA is connected to a high performance AXI slave interface which provides a high bandwidth data-path from the master modules (such as Master Output) in PL to the DDR on Zynq PS.

Laplacian Matrix Formation

To this end, Zynq PS has matrix B stored in the DDR memory. Matrix A can be read from the BRAM register address 0. Both matrices A and B are used to form the Laplacian matrix L on Zynq PS as in Algorithm 5. The matrix L size is expected to be $n-by-n$ where $n = 65536$ is NUM_VALUES. Hence, a total number of $65536x65536$ values will have to be streamed to the HLS SVD IP core to compute the eigenvalue and eigenvector decomposition and send back to the PS three matrices of size $n-by-n$ each. This comprises large utilization of BRAM resources used to store partial results of SVD (discussed in the next section). Consequently, the design may require to use BRAM resources more than the resource limit available on the Zynq-7000 platform. Thus, a work-around is to compute the eigenvalue and eigenvector of a smaller size matrix.

[37] found that if A is positive definite, then we can solve for the approximate eigenvectors in one step. Let $A^{\frac{1}{2}}$ denote the symmetric positive definite square root of A, then, define Q:

$$Q = A + A^{-\frac{1}{2}} B B^T A^{-\frac{1}{2}}, \tag{5.3}$$

and find its eigenvalue and eigenvector decomposition $Q = U \Lambda U^T$ where U is the eigenvector of Q and Λ contains the corresponding eigenvalues. Define a matrix V such that:

$$V = \begin{bmatrix} A \\ B^T \end{bmatrix} A^{-\frac{1}{2}} U \Lambda^{-\frac{1}{2}}. \tag{5.4}$$

[37] proved that the approximated similarity graph of the whole data, $\widehat{W}$, in Equation 2.26 is diagonalized by V and Λ such that $\widehat{W} = V \Lambda V^T$ where V is the eigenvector of $\widehat{W}$ and Λ contains the corresponding eigenvalues.

Accordingly, we can obtain the approximated eigenvalue and eigenvector decomposition of the whole HSI cube. Note that the size of Q is $(m-by-m = 128 \times 128)$ which is much smaller than $n-by-n$. Moreover, m random samples is a user defined parameter and can be chosen to be smaller than 128 as well.

To form the matrix Q, first, the inversion of A is obtained using Gauss-Jordan Elimination Algorithm in embedded C programming language (explained in Appendix B). Matrix B is also transposed to form $\mathbf{B^T}$.

Next, $\hat{d}$ of Equation 2.32 is calculated using $A\mathbf{1}_m$ and $B\mathbf{1}_n$, which are the row sum of matrices A and B and $B^T\mathbf{1}_m$ is the column sum of matrix B. $\mathbf{1}_m$ is a vector row of $\mathbf{1}$s of size m. A_{ij} and B_{ij} are accordingly updated as in Equations 2.33 and 2.34 given by Nystrom Extension. Finally, 3 matrix multiplications and 1 matrix addition are computed to form the matrix S.

The matrix Q is then streamed to the HLS SVD core using the address of the AXI_DMA_1 connected to the PL core in order to obtain its eigenvalue Λ and eigenvector U decomposition. This implementation differs from the Listing 3.6 where the SVD decomposition is found for the Laplacian matrix in the software implementation.

5.4 Eigenvalue and Eigenvector Decomposition

The HLS SVD IP core is used to obtain the eigenvalues and eigenvectors of matrix Q. Vivado HLS tool comes with C libraries that allow common hardware design constructs and functions to be easily modeled in C, and therefore, synthesized to RTL [46]. One of the popular libraries is the HLS Linear Algebra Library which provides a number of commonly used linear algebra functions including SVD using Two-sided Jacobi. In this work, a top C function is designed in HLS to stream in and out the input and outputs of the built-in SVD C function. In addition, the built-in function is modified to include user directives from Table 2.1 to achieve resource and throughput optimizations. These directives are included in special implementation controls within the HLS SVD function shown in Table 5.3 [46]. The HLS SVD function implements Algorithm 4 listed in Chapter 2. The top level function (svd_top) developed in this thesis is designed to

Table 5.3: SVD Implementation Controls.

Control	**Description**
Off-diagonal loop pipelining	Off-diagonal entries calculation loop pipelined. If > 4, enables Vivado HLS to further resource share and reduce the DSP utilization.
Diagonal loop pipelining	> 1: Enables Vivado HLS to resource share.
Iterations (NUM_SWEEPS)	Number of iterations (sweeps) for convergence. Literature typically suggests 6 to 10 iterations to successfully converge [46].

comply with AXI-stream master and slave interface and communicate with DMA. It synthesizes to the IP core shown in Figure 5.6. The input stream represents the matrix Q of size $ROWS \times COLS$ where $ROWS = COLS = m$. The output stream port is used to transfer the matrices S, V, and D one after the other. On the other hand, those matrices are collected and reconstructed by the Zynq PS.

The top level function is listed in Listing 5.1. The SVD controls can be configured using a class template, ***svd_traits*** and added as an argument to the ***svd_top*** function from the Linear Algebra Library as shown in Lines 1-4. AXI_VAL in Line 6 is a datatype which expresses the side channel information associated with AXI4-stream, namely <TDATA,TUSER,TID,TDEST>. TID, TDEST, and TUSER are all optional signals on the interface. Lines 11 - 16 specify the I/O interfaces to communicate with the SVD IP core. The input and output matrices are all assigned AXI_Stream interfaces. The control bus port is used for signaling whether or not the HLS core is ready to accept new data. Furthermore, this port is assigned AXI_Lite interface for configuration. Note that matrix $Q \longrightarrow A$ and matrix $U \longrightarrow D$ in the C code of the Listing 5.1 below.

Data is read from the streams as a means of packing/unpacking (poped/pushed) from the AXI_VAL stream datatype. Lines 22-27 shows data elements are poped from the input stream and saved in a 2D matrix to form the input A. Line 28 calls the built-in Vivado

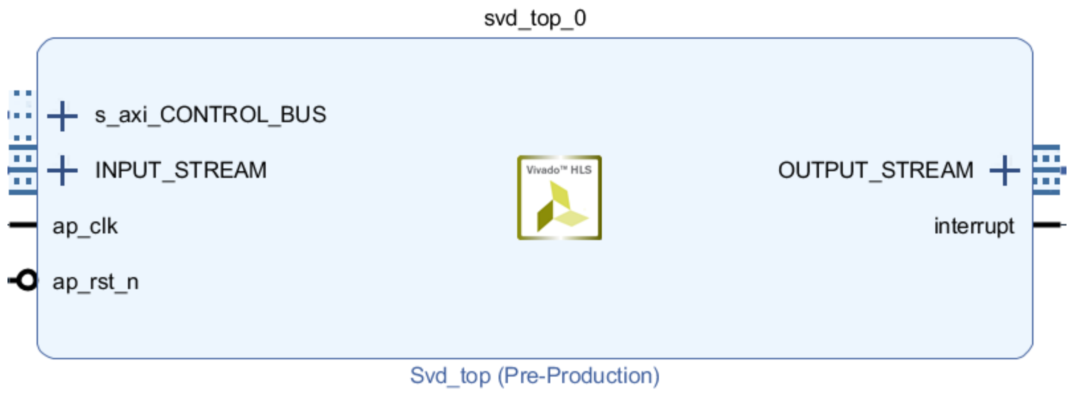


Figure 5.6: HLS SVD block design.

HLS SVD function with the user defined control configuration. The computed SUV data stored in memory (BRAM tiles when converted to RTL) is pushed to the output stream sequentially in Lines 30-47. Notice that a TLAST signal is concatenated with the last V element in Line 45. This signal is set high when $(k = ROWS * COLS * 3 - 1)$, in other words, the last computed element is streamed.

Code listing 5.1: HLS SVD C code.

```
1  struct MY_CONFIG : hls::svd_traits<A_ROWS,A_COLS,MATRIX_IN_T,MATRIX_OUT_T>{
2  static const int NUM_SWEEPS = 6;
3  static const int DIAG_II = 4;
4  static const int OFF_DIAG_II = 4;
5  };
6  typedef ap_axiu<32,1,1,1> AXI_VAL;
7  // The top-level function to synthesize
8  //
9  void svd_top (AXI_VAL INPUT_STREAM[A_ROWS*A_COLS], AXI_VAL OUTPUT_STREAM[A_ROWS*A_COLS*3])
10 {
11 #pragma HLS INTERFACE axis port=INPUT_STREAM
12 #pragma HLS INTERFACE axis port=OUTPUT_STREAM
13 // axi lite interface is used by the ARM processor to control the execution of this accelerator
14 // this line also creates an interrupt port for this module which later will be used by
15 // Zynq PS code to signal that data is ready.
16 #pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
17 MATRIX_IN_T  a_i[A_ROWS][A_COLS];
18 MATRIX_OUT_T s_i[A_ROWS][A_COLS];
19 MATRIX_OUT_T u_i[A_ROWS][A_ROWS];
20 MATRIX_OUT_T v_i[A_COLS][A_COLS];
21 // Copy input data to local memory
22 a_row_loop : for (int r=0;r<A_ROWS;r++) {
23     a_col_loop : for (int c=0;c<A_COLS;c++) {
24       int k = r*A_ROWS + c;
25       a_i[r][c] =  pop_stream<float,1,1,1>(INPUT_STREAM[k]);
26     }
27 }
28 hls::svd<A_ROWS, A_COLS, MATRIX_IN_T, MATRIX_OUT_T>(a_i, MY_CONFIG, s_i, u_i, v_i);
29 // Copy local memory contents to outputs
30 s_row_loop : for (int r=0;r<A_ROWS;r++) {
31  s_col_loop : for (int c=0;c<A_COLS;c++) {
32    int k = r*A_ROWS + c;
33    OUTPUT_STREAM[k] = push_stream<float, 1, 1, 1>(s_i[r][c]);
34  }
35 }
36 u_row_loop : for (int r=0;r<A_ROWS;r++) {
37  u_col_loop : for (int c=0;c<A_ROWS;c++) {
38         int k = r*A_ROWS + c + (A_ROWS*A_COLS);
39    OUTPUT_STREAM[k] = push_stream<float, 1, 1, 1>(u_i[r][c]);
40  }
41 }
42 v_row_loop : for (int r=0;r<A_COLS;r++) {
43  v_col_loop : for (int c=0;c<A_COLS;c++) {
44          int k = r*A_ROWS + c + (2*A_ROWS*A_COLS);
45    OUTPUT_STREAM[k] = push_stream<float, 1, 1, 1>(v_i[r][c], k == (A_ROWS*A_COLS*3 - 1));
46  }
47 }
48 }
```

The below Listing 5.2 describes the built-in (*svd*) function called from Listing 5.1 and is obtained from the HLS Library [46]. In this work, all directives from Table 2.1 are used to optimize the design.
The function is mainly controlled by the number of iterations for Steps 3 - 6 in Algorithm 4. Loop unrolling optimization is applied to the 3 loops going through the number of iterations, rows, and columns of A.
svd starts by fetching a_{pp}, a_{pq}, a_{qp}, and a_{qq} of Equation 2.10 in Chapter 2, termed as w, x, y, z, respectively in Lines 40-57. If it is the first iteration, then the values are read from the $\boldsymbol{A}$ matrix, otherwise, the values are fetched from ***s_in*** which is the transformed $\boldsymbol{A}$ from previous iteration. The function then calls ***svd***$2x2$ to carry out the calculation of the diagonal entries, a'_{pp} and a'_{qq}.

Code listing 5.2: HLS SVD built-in function.

```
1  void svd( const InputType A[RowsA][ColsA],
2                      class SVDTraits,
3                      OutputType S[RowsA][ColsA],
4                      OutputType U[RowsA][RowsA],
5                      OutputType V[ColsA][ColsA] )
6    {
7      // Assert that the matrix is symmetric
8  #ifndef __SYNTHESIS__
9      assert(RowsA==ColsA);
10 #endif
11
12     // Internal memories for partial results
13     typename SVDTraits::SIntType s_in[RowsA][ColsA];
14     typename SVDTraits::UIntType u_in[RowsA][ColsA];
15     typename SVDTraits::VIntType v_in[RowsA][ColsA];
16
17     // Current S,U,V values being worked on
18     typename SVDTraits::SIntType w_in, x_in, y_in, z_in;
19     typename SVDTraits::SIntType w_out, x_out, y_out, z_out;
20     typename SVDTraits::UIntType uw_in, ux_in, uy_in, uz_in;
21     typename SVDTraits::UIntType uw_out, ux_out, uy_out, uz_out;
22     typename SVDTraits::VIntType vw_in, vx_in, vy_in, vz_in;
23     typename SVDTraits::VIntType vw_out, vx_out, vy_out, vz_out;
24
25     // 2x2 Rotation values // c1 , s1, -s1, c1,     c2, s2, -s2, c2
26     typename SVDTraits::CSIntType uw_new, ux_new, uy_new, uz_new;
27     typename SVDTraits::CSIntType vw_new, vx_new, vy_new, vz_new;
28
29     sweep_loop: for(int sweepnum = 0; sweepnum < SVDTraits::NUM_SWEEPS; sweepnum++){
30       // NOTE: MIN_DIM = RowsA = ColsA
```

```
31       #pragma HLS UNROLL FACTOR = SVDTraits::DIAG_UNROLL_FACTOR
32       row_loop: for(int top_left = 0; top_left < SVDTraits::MIN_DIM; top_left++) {
33       #pragma HLS UNROLL FACTOR = SVDTraits::DIAG_UNROLL_FACTOR
34         col_loop: for(int bottom_right = top_left+1;
35         bottom_right< SVDTraits::MIN_DIM; bottom_right++) {
36         // loop unrolling and Pipelining
37         #pragma HLS UNROLL FACTOR = SVDTraits::DIAG_UNROLL_FACTOR
38         #pragma HLS PIPELINE II = SVDTraits::DIAG_II
39            // Fetch w,x,y,z values
40            if (sweepnum == 0 && top_left == 0) {
41              if (bottom_right == 1) {
42                w_in =A[top_left]    [top_left];
43                x_in =A[top_left]    [bottom_right];
44                y_in =A[bottom_right][top_left];
45              } else {
46                // Now revist values already updated in first diagonal pass
47                w_in =s_in[top_left]    [top_left];
48                x_in =s_in[top_left]    [bottom_right];
49                y_in =s_in[bottom_right][top_left];
50              }
51              z_in =A[bottom_right][bottom_right];
52            } else {
53              w_in =s_in[top_left]    [top_left];
54              x_in =s_in[top_left]    [bottom_right];
55              y_in =s_in[bottom_right][top_left];
56              z_in =s_in[bottom_right][bottom_right];
57            }
58
59            // Diagonal
60            svd2x2(w_in, x_in, y_in, z_in, uw_new, ux_new,
61            uy_new, uz_new, vw_new, vx_new, vy_new, vz_new,
62            w_out, x_out, y_out, z_out);
```

svd$2x2$ function is described in Listing 5.3. This function is inlined when being called by ***svd*** in the previous code listing. Lines 22-33 compute half angles θ_1 and θ_2 by making use of ***arctan*** for Equation 2.13. Next, Lines 36-61 carry out matrix multiplications which corresponds to Equations 2.15 and 2.16 by making use of an inlined function, ***vm***$2x1(a1, b1, a2, b2, c)$, which simply computes $c = a1 * b1 + a2 * b2$.

Code listing 5.3: *svd*$2x2$ function.

```
1  void svd2x2(
2      AInType  w_in,    AInType  x_in,    AInType  y_in,   AInType  z_in,
3      CSType   &uw_out, CSType   &ux_out, CSType &uy_out,  CSType   &uz_out,
4      CSType   &vw_out, CSType   &vx_out, CSType &vy_out,  CSType   &vz_out,
5      AOutType &w_out,  AOutType &x_out,  AOutType &y_out, AOutType &z_out)
6    {
7      // svd2x2 is inlined in into any functions calling svd2x2
8      #pragma HLS inline
9      const AOutType     outZERO = 0;
10     CSType             s1, c1, s2, c2;
11     AInType            u1, u2;
12     std::complex<AInType> A, B;
13     CSType             cosA_full, sinA_full, cosA_half, sinA_half;
14     CSType             cosB_full, sinB_full, cosB_half, sinB_half;
15     bool               A_is_pos_real, A_is_imag;
16     bool               B_is_pos_real, B_is_imag;
17     CSType             uw_int, ux_int, uy_int, uz_int;
18     CSType             vw_int, vx_int, vy_int, vz_int;
19     AOutType           w_out1, w_out2, z_out1, z_out2,
20                        w_out_int, z_out_int;
21     // Determine first half angle required to zero off-diagonal values
```

```
    u1 = z_in - w_in;  // theta1 - theta2
    u2 = y_in + x_in;  // theta1 - theta2
    A = u2/u1;
    calc_angle(A, cosA_full, sinA_full, cosA_half, sinA_half, A_is_pos_real,
    A_is_imag);

    // Determine second half angle, theta_1 + theta_2
    u1 = z_in + w_in;
    u2 = y_in - x_in;
    B = u2/u1;
    calc_angle(B,cosB_full, sinB_full, cosB_half, sinB_half, B_is_pos_real,
    B_is_imag);

    // Combine half angles to produce left and right rotations
    vm2x1(cosA_half,cosB_half,sinA_half,sinB_half,c1);  // c = a1*b1 + a2*b2;
    vm2x1(sinA_half,cosB_half,-cosA_half,sinB_half,s1);
    vm2x1(cosA_half,cosB_half,-sinA_half,sinB_half,c2);
    vm2x1(sinA_half,cosB_half,cosA_half,sinB_half,s2);
    // Build full U and V matrix
    uw_int = c1;
    ux_int = s1;
    uy_int = -s1;
    uz_int = c1;

    vw_int = c2;
    vx_int = s2;
    vy_int = -s2;
    vz_int = c2;
    // Apply rotation
    // - Uses the transpose version of U
    // w_out -> a'pp
    vm2x1(w_in,vw_int,x_in,vy_int,w_out1); // app.c2 - apq.s2
    vm2x1(y_in,vw_int,z_in,vy_int,w_out2); // aqp.c2 - aqq.s2
    vm2x1(uw_int,w_out1,uy_int,w_out2,w_out_int);
    // c1.(app.c2 - apq.s2) - s1.(aqp.c2 - aqq.s2)
    // z_out -> a'qq
    vm2x1(w_in,vx_int,x_in,vz_int,z_out1);
    vm2x1(y_in,vx_int,z_in,vz_int,z_out2);
    vm2x1(ux_int,z_out1,uz_int,z_out2,z_out_int);
    x_out = outZERO;
    y_out = outZERO;

    // Assign angle outputs
    uw_out = uw_int;
    ux_out = ux_int;
    uy_out = uy_int;
    uz_out = uz_int;
    vw_out = vw_int;
    vx_out = vx_int;
    vy_out = vy_int;
    vz_out = vz_int;
  }
```

The ***svd*** function then updates the diagonal entries of the matrices S, U, D (or SVD) using the outputs of ***svd***$2x2$ in Listing 5.4. Matrix S is updated in Lines 2-11 for the next iteration. Before updating U or V, Lines 16-32 assigns them to the identity matrix if they are the first entry, otherwise, Lines 44-50 perform 2 matrix multiplications, $Uout = U'.U$ and $Vout = V'.V$ where U' and V' are the new rotation angles obtained from ***svd***$2x2$, and U and V are the matrices from the previous iteration. The next iteration is then fetched.

Code listing 5.4: SVD top function continued 1.

```
// Update S on diagonal // s_in is updated for the next iteration
s_in[top_left]    [top_left]     = w_out;
s_in[top_left]    [bottom_right] = x_out;
s_in[bottom_right][top_left]     = y_out;
s_in[bottom_right][bottom_right] = z_out;
if (sweepnum == SVDTraits::NUM_SWEEPS-1) { // if last operation
 S[top_left]    [top_left]     = w_out;
 S[top_left]    [bottom_right] = x_out;
 S[bottom_right][top_left]     = y_out;
 S[bottom_right][bottom_right] = z_out;
}

// Update U & V
// o On the diagonal use a 2x2 as per the sigma
// o Need to create the identity matrix in U & V at the start
if (sweepnum == 0 && top_left == 0) {
 if (bottom_right==1) {
   uw_in = 1; // identity
   vw_in = 1;
 } else {
   // Now re-visiting diagonal values where u,v has been set
   uw_in = u_in[top_left][top_left];
   vw_in = v_in[top_left][top_left];
 }
 // identity matrix
 ux_in = 0;
 uy_in = 0;
 uz_in = 1;

 vx_in = 0;
 vy_in = 0;
 vz_in = 1;
} else { // previously updated U,V
 uw_in = u_in[top_left]    [top_left];
 ux_in = u_in[top_left]    [bottom_right];
 uy_in = u_in[bottom_right][top_left];
 uz_in = u_in[bottom_right][bottom_right];
 vw_in = v_in[top_left]    [top_left];
 vx_in = v_in[top_left]    [bottom_right];
 vy_in = v_in[bottom_right][top_left];
 vz_in = v_in[bottom_right][bottom_right];
}
// Update U, U_out = U'.U
mm2x2(uw_in, ux_in, uy_in, uz_in, uw_new, ux_new,
uy_new, uz_new, uw_out, ux_out, uy_out, uz_out);
// Update V, V_out = V'.V
mm2x2(vw_in, vx_in, vy_in, vz_in, vw_new, vx_new,
vy_new, vz_new, vw_out, vx_out, vy_out, vz_out);

// u_in, v_in for the next iteration.
u_in[top_left]    [top_left]     = uw_out;
u_in[top_left]    [bottom_right] = ux_out;
u_in[bottom_right][top_left]     = uy_out;
u_in[bottom_right][bottom_right] = uz_out;
v_in[top_left]    [top_left]     = vw_out;
v_in[top_left]    [bottom_right] = vx_out;
v_in[bottom_right][top_left]     = vy_out;
v_in[bottom_right][bottom_right] = vz_out;
if (sweepnum == SVDTraits::NUM_SWEEPS-1) {
 U[top_left]    [top_left]     = uw_out;
 U[top_left]    [bottom_right] = ux_out;
 U[bottom_right][top_left]     = uy_out;
 U[bottom_right][bottom_right] = uz_out;
 V[top_left]    [top_left]     = vw_out;
 V[top_left]    [bottom_right] = vx_out;
 V[bottom_right][top_left]     = vy_out;
 V[bottom_right][bottom_right] = vz_out;
}
```

Upon completion, matrices SUV are streamed back to Zynq PS where S contains the eigenvalues of $\boldsymbol{Q}$ and $U = V$ contain the eigenvectors.

5.5 Spectral Embedding and K-means Clustering

Matrix V of Equation 5.4 is formed as means of 3 matrix multiplications involving the eigenvector U and the square root inverse of eigenvalues matrix Λ in Zynq PS.
According to Algorithm 5, the next step at this stage is to carry out spectral embedding. Lines 41-48 of Listing 3.6 in Chapter 3, are implemented in C for spectral embedding in Listing 5.5 below. Lines 3-9 compute the spectral embedding of the eigenvectors using the 1st eigenvector ($V[row][0]$). A built-in C function $qsort()$ is used in Line 19 for sorting in descending order, the eigenvalues of Q obtained from the HLS output stream. Qsort makes use of an implemented function, $compare$, which takes two void pointers as arguments and returns their difference; this is how $qsort()$ determines which eigenvalue is smaller, or larger, than the other. Lines 20-26 comprises the choice of the first k sorted eigenvectors where k is the number of clusters, the size of $embed_{mask}$ is $n - by - k$ as in Algorithm 5.

Code listing 5.5: Spectral Embedding in Zynq PS.

```
1  float embed [num_values][m_samples];
2  float embed_mask [num_values][k];
3  for (int col = 0 ; col<k ; col++)
4  {
5      for (int row = 0 ; row<num_values ; row++)
6      {
7          embed[row][col] = V_Mat[row][col+1]/V_Mat[row][0];
8      }
9  }
10 struct EigenValues { // storing eigenvalues of S into a struct
11     int index;
12     int eigenvalue;
13 };
14 struct EigenValues eigen[num_values];
15 for(int i=0 ; i<num_values ; i++) {
16   eigen[i].index = i;
17   eigen[i].eigenvalue = Q[i];
18 }
19 qsort (eigen, num_values, sizeof(struct EigenValues), compare); // std C quicksort function
20 for (int col = 0 ; col<k ; col++)
21 {
22     for (int row = 0 ; row<num_values ; row++)
23     {
24     embed_mask [row][col] = embed[row][eigen.index[col]]; // first k sorted eigenvectors
25     }
26 }
```

The next step is to formulate the vector y of Algorithm 5 with size k as illustrated in Figure 5.7. This is done by constructing a vector pointer $y[i]$, corresponding to the i-th row of E.

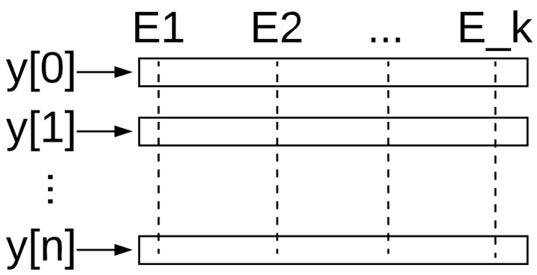


Figure 5.7: y pointer array points to the rows of E.

The implementation of k-means clustering follows the same steps as in Algorithm 2 of the standard k-means. Before clustering data, a random number of data points (total k) are selected as centers to form the initial cluster centroids. This is done in Lines 1-11 of Listing 5.6. A centroid is defined as a row $y[i]$ with size k. In other words, initially, the algorithm randomly samples k centroids of the dimension size k from the list y.

Code listing 5.6: K-means Clustering function call.

```
1   srand((unsigned) 0);
2   point_num = rand();
3   dim = k;
4   float *centroids = (float *)malloc(sizeof(float) * dim * k);
5   for ( int clust_num = 0; clust_num < k; clust_num++ )
6   {
7       point_num = rand();
8       point_num = point_num % n;
9       for ( int dim_num = 0; dim_num < dim; dim_num++ )
10      {
11      centroids[clust_num*dim + dim_num] = y[point_num*dim + dim_num];
12      }
13  }
14  kmeans(dim,                  // dimension of data -> k columns of E
15           y, // 1D pointer to eigenvectos
16         n,           // number of elements of y.
17           k,           // number of clusters
18           centroids,         // initial cluster centroids
19           final_cluster_assignment  // output
20           );
```

Listing 5.7 lists the k-means clustering implemented function. K-means clustering runs for a number of iterations ($MAX\_ITERATIONS = 100$). The function starts by calculating the distances between all the points and all the centroids, results for each vector point y is saved in $dist$ in Line 8. According to the distances calculated, clusters are reassigned to the data points as well as cluster assignment holder is updated in Line 9. This constitutes one iteration run. The same process is repeated for a specified number of iterations. The C code developed is available on [50].

Code listing 5.7: K-means Clustering function.

```
1   void kmeans ( int dim, float*y, int n, int k, float *centroids,
2   int *final_cluster_assignment)
```

```c
{
int MAX_ITERATIONS = 100;
float *dist                   = (float *)malloc(sizeof(float) * n * k);
int   *cluster_assignment_cur  = (int *)malloc(sizeof(int) * n);
// initial setup
calc_all_distances(dim, n, k, y, cluster_centroid, dist);
assign_all_clusters_from_distances(dim, n, k, dist, cluster_assignment_cur);
// update cluster centroids
calc_cluster_centroids(dim, n, k, y, cluster_assignment_cur, cluster_centroid);
int i = 1;
while (i < MAX_ITERATIONS)
 {
 // move all points to nearest cluster
 calc_all_distances(dim, n, k, y, cluster_centroid, dist);
 assign_all_clusters_from_distances(dim, n, k, dist, cluster_assignment_cur);
 calc_cluster_centroids(dim, n, k, y, cluster_assignment_cur, cluster_centroid);
 i++;
 }
}
```

Chapter 6

HW/SW Co-design Results

This chapter presents results of HW/SW co-design of spectral clustering algorithm developed for Zynq-7000 SoC (in Chapter 5). The results are divided into three sections, namely, HW/SW performance analysis, resource utilization, and classification performance analysis. Performance analysis and resource utilization discussion are based on post-synthesis results of the generic values of Table 5.1

6.1 Performance Analysis

HW/SW co-design performance analysis are based on the generic values set for the Salinas dataset. The original dataset is initially pre-processed by MATLAB to reduce its dimensions to 16 spectral bands by making use of PCA. Furthermore, the HSI cube is saved in an SD card as a binary file.

Table 6.1 shows the measured execution time of the spectral clustering algorithm on Salinas-A dataset. The HW/SW co-design is tested on Zedboard Development board containing Zynq-7000 SoC with -O3 compiler directive for performance optimization. The speed-up is evaluated in comparison with the software solution execution time as reported by FSC [12] running on Intel(R) Core(TM) i7-5930K CPU, 3.50 GHz, 64 GB memory, Ubuntu 14.04.5 LTS system. The HW/SW co-design achieves a speed-up factor

Table 6.1: Performance comparison for HW/SW codesign solution.

Implementation	Execution Time	Speed-up Factor
FSC Intel i7-5930K [12]	**4.62 s**	**1.723**
HW/SW design (100MHz)	**2.68 s**	1
Graph Construction	0.75 s	
HLS SVD (with optimizations)	0.67 s	
HLS SVD (without optimizations)	2.98 s	
Zynq PS operations	1.26 s	

of **1.723** when clocked with frequency of 100 MHz compared to Fast Spectral Clustering solution. In addition, the table shows individual time taken for Graph Construction

of A and B module and the HLS SVD module. Hardware timings are measured using AXI_TIMER and maximum clock latency/clock cycles. It can be observed that the HLS SVD core is the most expensive module in time complexity. This is because of the large matrix data that is decomposed into 3 other matrices of the same size. Furthermore, time has been measured with and without the user directive, resource and throughput, optimizations of Tables 2.1 and 5.3. The optimization owed to a speed-up factor of **4.45** for the HLS SVD core performance.
Zynq PS operations time includes the total time taken for HSI read from SD, two data transfers, Q and V matrix formations, spectral embedding, and k-means clustering. Software time is measured using C XTime library.

The highest achievable operating frequencies for modules synthesized on XC7Z020-CLG484-1 SoC are shown in Table 6.2. Each hardware module in the design have been individually analyzed to determine the maximum operating frequencies by making use of the timing reports.

Table 6.2: Maximum frequency for HW/SW codesign solution modules.

Module	Minimum Time Period	Maximum Frequency
PS-PL System	9.955 ns	100.45 MHz
Graph Construction	6.270 ns	159.48 MHz
HLS SVD	6.343 ns	157.63 MHz
Master Output	2.362 ns	423.45 MHz
BRAM Module	3.363 ns	297.35 MHz

6.2 Resource Utilization

Post-synthesis resource utilization for HW/SW co-design using the generic parameters is shown in Table 6.3. The table shows the total resources used and their percentage of the total resources available on xc7z020clg484-1 SoC. HSI data input saved in BRAM module is synthesized to use 64 BRAM tiles which accounts for approximately for 45.7% of the available BRAM on the SoC device. As for HLS SVD core, the used BRAM tiles accounts for 16.4% of the total BRAM tiles available.

The high use of BRAM tiles (77.14% with 16 spectral bands) can be a limitation for this HW/SW co-design solution. Figure 6.1 shows the *BRAM module* resource utilization of BRAM tiles, slice LUTs, and slice registers as a function of the number of spectral bands varrying from 16 to 32 in steps of 4. The limitation can be observed at 28 spectral bands where the total number of synthesized BRAM tiles (BRAM module + PL) goes beyond the available resources (140 BRAM tiles). This is because as the number of spectral bands increases, the size of the HSI cube increases as well. Consequently, the BRAM_ADDR_WIDTH parameter increases and more BRAM rows are required by the design. However, the design meets the requirements outlined in section 1.2 of operating on at least 20 spectral bands. In addition, as expected, slice LUTs and registers slightly increase with the number of spectral bands.

Table 6.3: Post-Synthesis resource utilization report.

Module	Slice LUTs	Slice Registers	DSP Blocks	BRAM Tiles
PS-PL system	**5066**	**6285**	**0**	**0**
Input logic	**1657**	**2293**	**0**	**82**
AXI_DMA_1	1504	1975	0	2
BRAM module	153	318	0	80
Processing logic	**15466**	**10558**	**41**	**25**
Graph A and B	532	900	0	1
AXI_DMA_2	1504	1975	0	2
HLS SVD	13356	7603	41	23
Master Output	74	80	0	0
Total	**22189**	**19056**	**41**	**108**
Utilization %	**41.71%**	**17.98%**	**18.64%**	**77.14%**

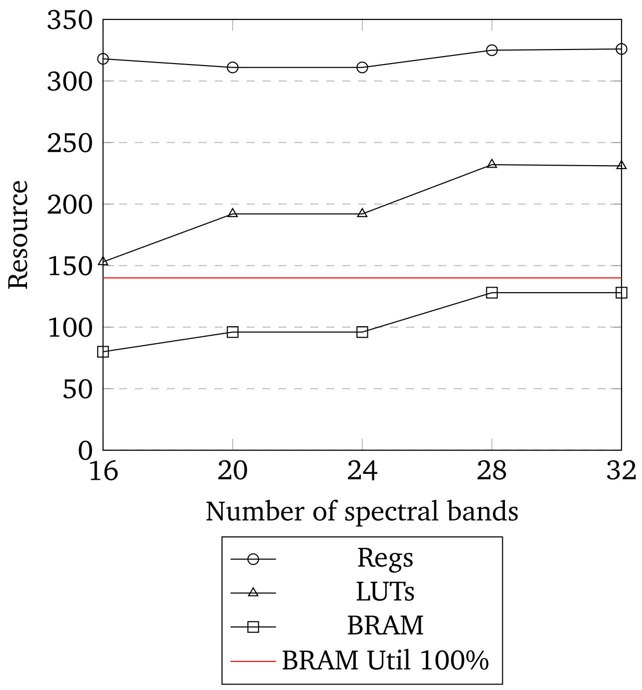


Figure 6.1: Resource utilization as a function of varying number of spectral bands for BRAM module.

Processing logic designed in section 5.3 make use of the same resources in a pipelined design approach rather than duplicating the use of resources and therefore, a parallel design approach. The main idea is to share the resources between each stage of processing. An example is the register used to hold *X_i*, 1 register is used to save the *X_i* and that same register is reused for the entire computation process. In addition, the size of the streaming packet in *Master Output* module is kept the same throughout the variation of HSI size since the BRAM_DATA_WIDTH is the same. This is applicable in matrix *B* where the size of the matrix increases greatly with the change of number of spectral bands but it does not have an effect on the resource utilization since the matrix data width size is not changed. The main constraint however, is on the time taken and maximum operating frequency. On the other hand, HLS SVD core depends on the size of matrix Q. In this case, Q's size is kept constant at $m - by - m = 128 \times 128$ throughout the experiments. This also leads to having a constant BRAM tiles usage size of 25 as shown in Figure 6.2. Hence, the resource utilization for the PL solution changes slightly or remains constant as the number of spectral bands increases. This can be observed in Figure 6.2.

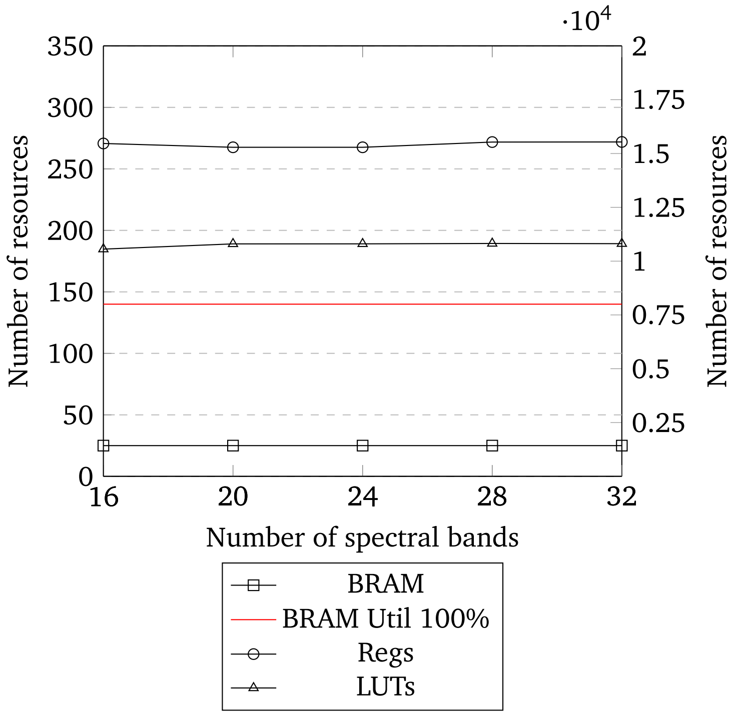


Figure 6.2: Resource utilization as a function of varying number of spectral bands for Processing Logic modules.

6.3 Clustering Performance Analysis

The HW/SW co-design implementation of spectral clustering algorithm is experimented on 4 different datasets, 2 of which are water-based datasets for the purpose of HYPSO mission. Experimental datasets include Urban, Jasper Ridge, Samson, and Salinas-A. Experiment results are compared with 2 other clustering methods including the Clustering using Binary Partition Trees (CLUS-BPT) developed in this work, and Fast Spectral Clustering (FSC)[12]. Furthermore, the experiments are carried out on the final produced k-means clusters of each dataset for HW/SW co-design solution. The produced output is saved as a binary file on the SD card and is further used for analysis. Moreover, the classification labels are plotted as well as evaluated using NMI and Purity metric scores on Python 3.7.3. The number of clusters for HW/SW solution are fixed according to the corresponding number of classes of each dataset obtained from section 4.2.

Table 6.4 shows the highest purity and NMI scores obtained by the compared methods for Urban, Jasper Ridge, Samson, and Salinas-A datasets. Figures 6.3, 6.4, 6.5, and 6.6 present the final cluster maps analogous to the valued mentioned in Table 6.4 for each method. Note that the image results of FSC are horizontally flipped for comparison reasons. Generally, cluster color labels may not match the ground truth color reference because clustering methods do not learn the exact labels (supervised learning) but rather assigns them. For example, label #1 in one cluster map may correspond to label #7 in the ground reference image. The following observations are based on the aforementioned table and figures results:

Table 6.4: Best purity and NMI scores obtained by HW/SW co-design proposed method, FSC SW, and CLUS-BPT SW for images.

Framework / Dataset	FSC (SW) [12]			Proposed CLUS-BPT (SW)			Proposed HW/SW		
	Purity	NMI	k	Purity	NMI	k	Purity	NMI	k
Urban	0.51	0.21	4	0.90	0.2906	4	**0.8325**	**0.62**	4
Jasper Ridge	0.83	0.71	4	0.7652	0.5658	4	**0.8043**	**0.54**	4
Samson	0.85	0.75	3	0.6896	0.6698	3	**0.8653**	**0.793**	3
Salinas-A	0.80	0.81	6	0.8753	0.8572	6	**0.5528**	**0.6562**	6

- Table 6.4 shows that the HW/SW solution resulted in significantly better values for Urban image than the other methods. The high normalized mutual information (NMI) score for those images shows that the percentage of how much information is mutual (or common) between the clustered labels and the truth labels is high. This can be observed further on Figure 6.3, the proposed HW/SW design was able to identify all the class labels for Urban image, unlike CLUS-BPT which completely misclassified the roof as asphalt. On the other hand, the software version of spectral clustering (FSC) clustered the image with more than 4 clusters and the comparison is inapplicable.
- As for Jasper-Ridge image, the proposed HW/SW solution resulted in comparable scores to the ones obtained by CLUS-BPT using 4 clusters. In contrast, FSC scored better than the proposed methods and this can be observed in Figure 6.4 where the cluster map (a) had fewer classification errors than the cluster maps (b) and (c). Moreover, the HW/SW cluster map misclassified a portion of the water class as road while CLUS-BPT had some classification errors spread across the image.
- HW/SW solution scored another significant results with the Samson image. The resulted purity and NMI socre are higher than FSC and CLUS-BPT. The cluster map of the proposed solution had the least number of misclassification errors and is very close to the ground truth labels compared to the other two methods. Moreover, FSC had some soil labels classified as Trees. However, all methods successfully classified the water label.
- Both FSC and CLUS-BPT performed significantly better for Salinas-A as compared to the proposed HW/SW co-design. The Salinas image in Figure 6.6 (d) contains 6 classes, all of which are well-localized spatially. However, it is a challenging image since there is a high probability of mixing true clusters together. For example, on the lower right of the image, all methods mix together the corn green weeds and Lettuce romaine 7 weeks labels without any background label separation.

However, FSC and CLUS-BPT managed to separate distinct labels for the rest of the image while the proposed HW/SW solution struggled in partially identifying the lettuce romaine 5 weeks and mistakened it with lettuce romaine 6 weeks.

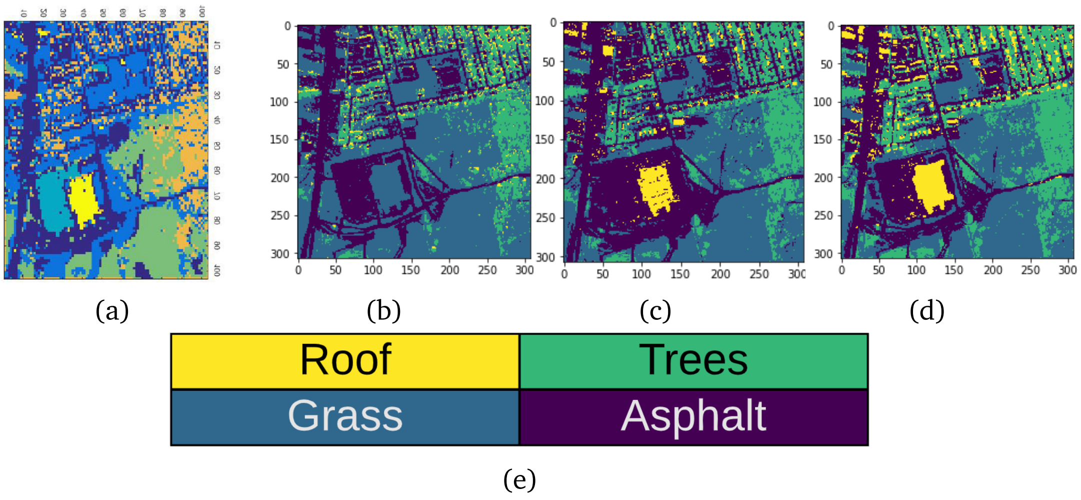


Figure 6.3: Cluster maps obtained from (a) FSC [12], (b) proposed CLUS-BPT, (c) proposed HW/SW, (d) ground reference image, and (e) ground reference color codes of classes for Urban image.

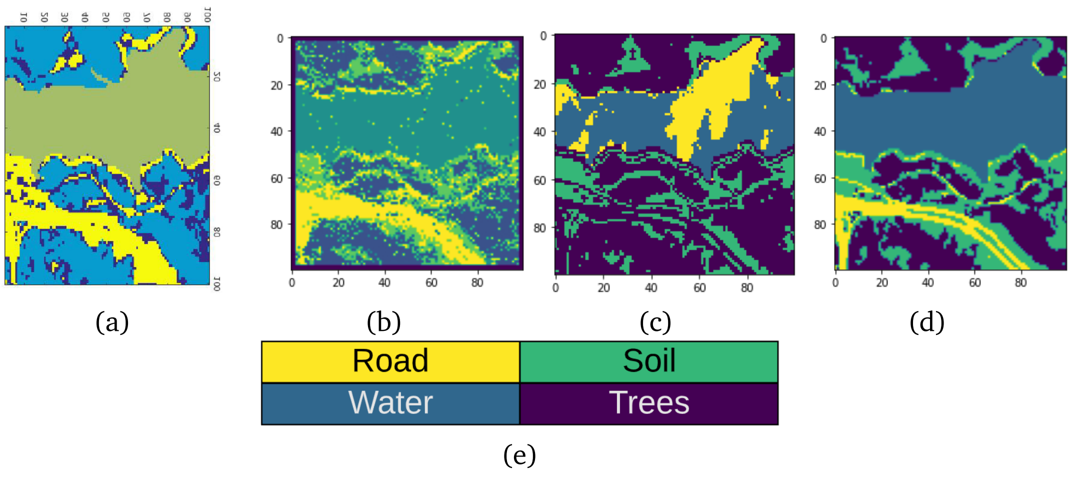


Figure 6.4: Cluster maps obtained from (a) FSC [12], (b) proposed CLUS-BPT, (c) proposed HW/SW, (d) ground reference image, and (e) ground reference color codes of classes for Jasper Ridge image.

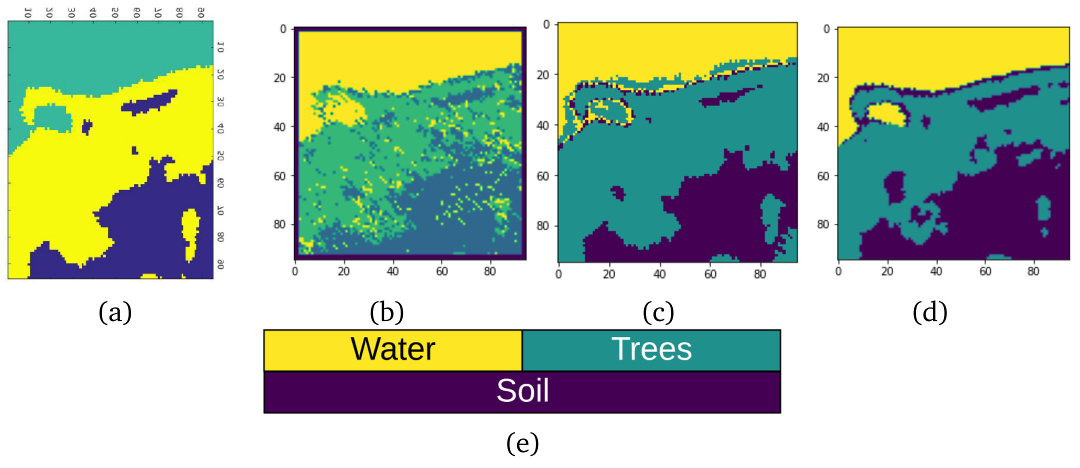


Figure 6.5: Cluster maps obtained from (a) FSC [12], (b) proposed CLUS-BPT, (c) proposed HW/SW, (d) ground reference image, and (e) ground reference color codes of classes for Samson image.

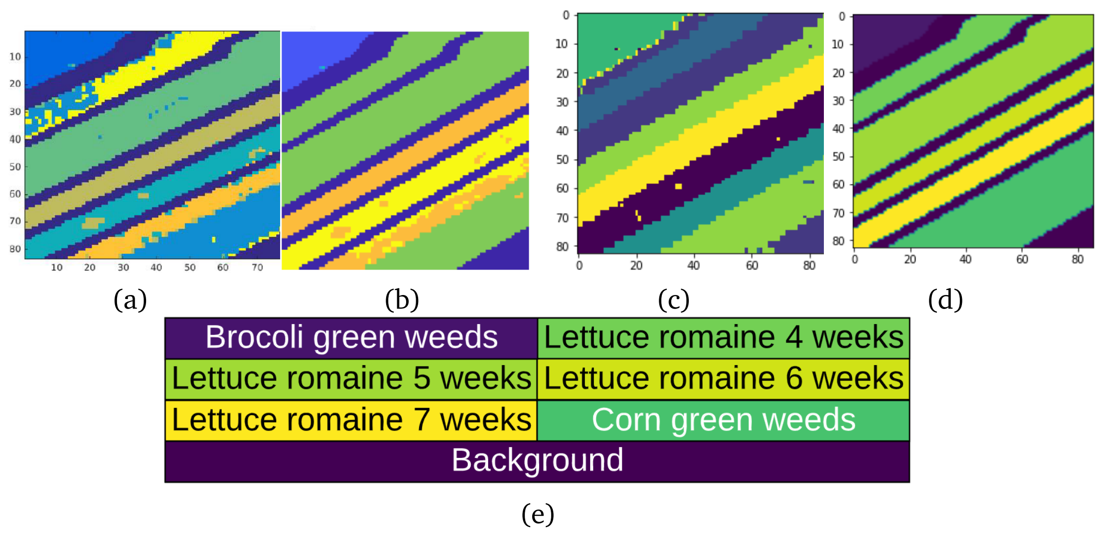


Figure 6.6: Cluster maps obtained from (a) FSC [12], (b) proposed CLUS-BPT, (c) proposed HW/SW, (d) ground reference image, and (e) ground reference color codes of classes for Salinas-A image.

Chapter 7

Conclusion

In this thesis, unsupervised clustering classification for hypersepctral imagery has been presented as a very promising area of research [4]. Two solutions were proposed, namely, a software framework of filtering k-means clustering based on binary partition trees segmentation, and a hardware/software co-design implementation of spectral clustering using Nyström Extension. The two algorithm implementations have been profiled on software for performance including classification evaluation scores and computational time. Furthermore, both software designs were tested on 8 widely used hyperspectral image datasets. The results of the experimented designs serve as a solid ground for the HW/SW solution implementation. The software versions of the two algorithms are developed using MATLAB R2019a and Python 3.7.3, respectively. On the other hand, the HW/SW co-design made use of a number of different development tools including Xilinx Vivado Design Suite, Vivado HLS, Xilinx SDK, and MathWorks HDL Coder™.

Unsupervised clustering classification has been preferred in this work to other classification methods since clustering succeeds in: 1) identifying the number of class labels in an image regardless of how many/what classes is the data divided into and 2) being based on linear algebra computations which are very feasible to be implemented on FPGAs.

For CLUS-BPT, the framework made use of segmentation based on binary partition trees. This tree-based structure has been preferred in a hyperspectral image context since BPT proved to be accurately presenting: 1) the decomposition of the image in terms of regions representing significant parts of the hyperspectral image and 2) the inclusion relations of the regions in the scene [9]. Furthermore, given its fixed structure, BPT allows implementing efficient and advanced application-dependent techniques on it including k-means clustering algorithm. The k-means clustering is based on a filtering algorithm that applies k-means clustering on tree-based data structures. Principal component analysis is integrated in the design to get the maximum data variance direction which is embedded with the final segmentation map in a new clustering framework.

Spectral clustering using Nyström Extension software algorithm was adapted from [37] and its implementation on hyperspectral images was inspired from [12]. Moreover, the affinity matrix of the entire HSI cube is efficiently approximated using Nyström Extension and is applicable to large-scale hyperspectral image datasets. The HW/SW solution made use of the algorithm presented in [37] where eigenvalue decomposition of the Laplacian matrix represented a major challenge for this method. Instead, the eigenvalue decomposition is found for a smaller size matrix using singular valued decomposition (SVD). A more optimized high-level synthesis of singular value decomposition is implemented using Vivado HLS. HW/SW cluster label results presented challenging evaluation scores as compared to other clustering methods. Moreover, the HW/SW solution maintains or even improves the clustering quality of the software version in 3 out of 4 experimented HSI datasets.

7.1 Future Work

Both designs of the proposed methods are divided into a number of different components and stages that can be modified or replaced with other applicable parts. This creates more room for future development and experiments, some of which are summarized below:

- For CLUS-BPT, dimensionality reduction can be done using methods other than PCA such as Linear Discriminant Analysis (LDA) and Non-negative matrix factorization (NMF). Both methods are used in pattern recognition context for feature extraction. In addition, NMF is able to preserve more information than PCA as demonstrated by [51]. A further research can be done on the effect of number of PCA components of a reduced HSI cube on the clustering performance.
- The HW/SW co-design takes advantage of the inherent parallelism in the design implementation by using the pipeline approach. Hence, less hardware resources are used but this trades-off the execution time. Accordingly, HW/SW design can be intensively parallelized at many points including graph construction of matrices A and B. In addition, further testing is necessary on different HSI datasets to define possible improvements and limitations.
- Full FPGA implementation is possible for the software computations of the HW/SW solution. This is because there exists in literature a number of FPGA implementations for matrix multiplications and k-means clustering. However, the design could be limited by the hardware resources available to use on-board.

Bibliography

[1] S. Watson, B. Whitton, S. Higgins, H. Paerl, B. Brooks and J. Wehr, 'Harmful algal blooms', in. Jun. 2015, pp. 873–920, ISBN: 9780123858764. DOI: 10.1016/B978-0-12-385876-4.00020-7.

[2] *The algae bloom in northern norway*. [Online]. Available: https://www.kontali.no/references/fhf_901574.

[3] Mariusz E. Grøtte, Roger Birkeland, Joao F. Fortuna, Julian Veisdal, Milica Orlandic, Evelyn Honore-Livermore, Gara Quintana-Diaz, Harald Martens, J. Tommy Gravdahl,, Fred Sigernes, Jan Otto Reberg, Geir Johnsen, Kanna Rajan, and Tor A. Johansen, 'Hyperspectral imaging small satellite in multi-agent marine observation system', *Unpublished-Internal document*, 2018.

[4] D. Chutia, D. K. Bhattacharyya, K. K. Sarma, R. Kalita and S. Sudhakar, 'Hyperspectral remote sensing classifications: A perspective survey', *Transactions in GIS*, Aug. 2015. DOI: 10.1111/tgis.12164.

[5] A. Signoroni, M. Savardi, A. Baronio and S. Benini, 'Deep learning meets hyperspectral image analysis: A multidisciplinary review', *Journal of Imaging*, vol. 5, p. 52, May 2019. DOI: 10.3390/jimaging5050052.

[6] M. Ismail, 'Unsupervised clustering for hyperspectral image classification', 2019.

[7] J. Lei, L. Wu, Y. Li, W. Xie, C.-I. Chang, J. Zhang and B. Huang, 'A novel fpga-based architecture for fast automatic target detection in hyperspectral images', *Remote Sensing*, vol. 11, p. 146, Jan. 2019. DOI: 10.3390/rs11020146.

[8] D. Lavenier, 'Fpga implementation of the k-means clustering algorithm for hyperspectral images', Sep. 2000.

[9] S. Valero, P. Salembier and J. Chanussot, 'Hyperspectral image representation and processing with binary partition trees', *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 22, Dec. 2012. DOI: 10.1109/TIP.2012.2231687.

[10] *Cs231n: Convolutional neural networks for visual recognition*, class notes for Neural Networks, Stanford, CA: Stanford Vision and Learning Lab, 2020. [Online]. Available: https://cs231n.github.io/neural-networks-1/.

[11] S. Roy, G. Krishna, S. R. Dubey and B. Chaudhuri, 'Hybridsn: Exploring 3d-2d cnn feature hierarchy for hyperspectral image classification', Feb. 2019.

[12] Y. Zhao, Y. Yuan and Q. Wang, 'Fast spectral clustering for unsupervised hyperspectral image classification', *Remote Sensing*, vol. 11, p. 399, Feb. 2019. DOI: 10.3390/rs11040399.

[13] S. Ranjan, D. Nayak, S. Kumar, R. Dash and B. Majhi, 'Hyperspectral image classification: A k-means clustering based approach', pp. 1–7, Jan. 2017. DOI: 10.1109/ICACCS.2017.8014707.

[14] A. Mehta and O. Dikshit, 'Segmentation-based clustering of hyperspectral images using local band selection', *Journal of Applied Remote Sensing*, vol. 11, p. 015 028, Mar. 2017. DOI: 10.1117/1.JRS.11.015028.

[15] Y. Tarabalka, J. Benediktsson and J. Chanussot, 'Spectral–spatial classification of hyperspectral imagery based on partitional clustering techniques', *Geoscience and Remote Sensing, IEEE Transactions on*, vol. 47, pp. 2973–2987, Sep. 2009. DOI: 10.1109/TGRS.2009.2016214.

[16] V. Dey, Y. Zhang and M. Zhong, 'A review on image segmentation techniques with remote sensing perspective', *ISPRS TC VII Symposium - 100 Years ISPRS*, vol. 38, Jan. 2010.

[17] A. Mehta and O. Dikshit, 'Projected clustering of hyperspectral imagery using region merging', *Remote Sensing Letters*, vol. 7, pp. 721–730, Aug. 2016. DOI: 10.1080/2150704X.2016.1182661.

[18] A. Mehta and O. Dikshit, 'Segmentation-based projected clustering of hyperspectral images using mutual nearest neighbour', *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. PP, pp. 1–8, Nov. 2017. DOI: 10.1109/JSTARS.2017.2768079.

[19] A. Mehta, A. Ashapure and O. Dikshit, 'Segmentation based classification of hyperspectral imagery using projected and correlation clustering techniques', *Geocarto International*, vol. 31, pp. 1–28, Oct. 2015. DOI: 10.1080/10106049.2015.1110207.

[20] C. Aggarwal, C. Procopiuc, J. Wolf, P. Yu and J. Park, 'Fast algorithms for projected clustering', vol. 28, Jun. 1999, pp. 61–72. DOI: 10.1145/304181.304188.

[21] F. Winterstein, S. Bayliss and G. A. Constantinides, 'Fpga-based k-means clustering using tree-based data structures', *23rd International Conference on Field programmable Logic and Applications*, 2013. [Online]. Available: https://ieeexplore.ieee.org/document/6645501/citations#citations.

[22] H. Futchs, Z. Kedem and B. Naylor, 'On visible surface generation by a priori tree structures', vol. 14, Dec. 1988, pp. 39–48. DOI: 10.1145/800250.807481.

[23] Y. Tarabalka, J. Chanussot and J. Benediktsson, 'Segmentation and classification of hyperspectral images using watershed transformation', *Pattern Recognition*, vol. 43, pp. 2367–2379, Jul. 2010. DOI: 10.1016/j.patcog.2010.01.016.

[24] S. Beucher and F. Meyer, 'The morphological approach to segmentation: The watershed transformation', in. Jan. 1993, vol. Vol. 34, pp. 433–481, ISBN: 9781315214610. DOI: 10.1201/9781482277234-12.

[25] G. Noyel, J. Angulo and D. Jeulin, ‘Morphological segmentation of hyperspectral images’, *Image Analysis and Stereology*, Nov. 2007. DOI: 10.5566/ias.v26.p101-109.

[26] M. Veganzones, G. Tochon, M. Dalla Mura, A. Plaza and J. Chanussot, ‘Hyperspectral image segmentation using a new spectral unmixing-based binary partition tree representation’, *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 23, Jun. 2014. DOI: 10.1109/TIP.2014.2329767.

[27] S. Valero, P. Salembier and J. Chanussot, ‘Comparison of merging orders and pruning strategies for binary partition tree in hyperspectral data’, Oct. 2010, pp. 2565–2568. DOI: 10.1109/ICIP.2010.5652595.

[28] S. Valero, P. Salembier, J. Chanussot and C. Cuadras, ‘Improved binary partition tree construction for hyperspectral images: Application to object detection’, Aug. 2011, pp. 2515–2518. DOI: 10.1109/IGARSS.2011.6049723.

[29] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman and A. Wu, ‘An efficient k-means clustering algorithm analysis and implementation’, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, pp. 881–892, Jul. 2002. DOI: 10.1109/TPAMI.2002.1017616.

[30] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman and A. Wu, ‘An efficient k-means clustering algorithm analysis and implementation’, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, pp. 881–892, Jul. 2002. DOI: 10.1109/TPAMI.2002.1017616.

[31] U. Luxburg, ‘A tutorial on spectral clustering’, *Statistics and Computing*, vol. 17, pp. 395–416, Jan. 2004. DOI: 10.1007/s11222-007-9033-z.

[32] J. Shi and J. Malik, ‘Normalized cuts and image segmentation’, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, May 2002. DOI: 10.1109/34.868688.

[33] R. Bhatia, *Positive Definite Matrices*. Princeton University Press, 2007.

[34] N. J. Salkind, *Encyclopedia of measurement and statistics*. Sage, 2007.

[35] G. Strang, *Introduction to Linear Algebra*, Fourth. Wellesley, MA: Wellesley-Cambridge Press, 2009, ISBN: 9780980232714.

[36] M. Berry, D. Mezher, B. Philippe and A. Sameh, ‘Parallel algorithms for the singular value decomposition’, in. Jan. 2005, pp. 117–164.

[37] C. Fowlkes, S. Belongie, F. Chung and J. Malik, ‘Spectral grouping using the nystrom method’, *IEEE transactions on pattern analysis and machine intelligence*, vol. 26, pp. 214–25, Mar. 2004. DOI: 10.1109/TPAMI.2004.1262185.

[38] Y. Zhao, Y. Yuan, F. Nie and Q. Wang, ‘Spectral clustering based on iterative optimization for large-scale and high-dimensional data’, *Neurocomputing*, vol. 318, Sep. 2018. DOI: 10.1016/j.neucom.2018.08.059.

[39] C. Fowlkes, S. Belongie and J. Malik, ‘Efficient spatiotemporal grouping using the nystrom method’, vol. 1, Feb. 2001, pp. I–231, ISBN: 0-7695-1272-0. DOI: 10.1109/CVPR.2001.990481.

[40] *Zynq-7000 soc data sheet*, Xilinx, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[41] *7 series dsp48e1 slice*, Xilinx, 2018. [Online]. Available: https://www.xilinx.com/support/%20documentation/user_guides/ug479_7Series_DSP48E1.pdf.

[42] *Axi4-lite specification*, ARM, 2010. [Online]. Available: https://static.docs.arm.com/ihi0051/a/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf?_ga=2.242520138.971183140.1590610019-178701575.1590610019.

[43] *Axi4-stream specification*, ARM, 2010. [Online]. Available: https://static.docs.arm.com/ihi0051/a/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf.

[44] *Axi dma v7.1*, Xilinx, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[45] *7 series fpgas memory resources*, Xilinx, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[46] *Vivado high-level synthesis*, Xilinx, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf.

[47] (2018). Hyperspectral remote sensing scenes, [Online]. Available: http://lesun.weebly.com/hyperspectral-data-set.html.

[48] Đ. Bovsković, M. Orlandić, S. Bakken and T. A. Johansen, 'Hw/sw implementation of hyperspectral target detection algorithm', *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–6, 2019.

[49] *Hdl coder user's guide*, MathWorks, 2020. [Online]. Available: https://ww2.mathworks.cn/help/pdf_doc/hdlcoder/hdlcoder_ug.pdf.

[50] M. Ismail, https://https://github.com/mh3081995/HW-SW-HSI-Classification, 2020.

[51] B. Rén, L. Pueyo, G. Zhu, J. Debes and G. Duchene, 'Non-negative matrix factorization: Robust extraction of extended structures', *The Astrophysical Journal*, vol. 852, Dec. 2017. DOI: 10.3847/1538-4357/aaa1f2.

[52] R. Bro and A. Smilde, 'Principal component analysis', *Analytical methods*, vol. 6, p. 2812, May 2014. DOI: 10.1039/c3ay41907j.

Appendix A

Dimensionality Reduction

Principal component analysis (PCA) is a useful statistical technique that has found application in fields such as face recognition and image compression, and is a common technique for finding patterns in data of high dimension, such as hyperspectral data. In this thesis, PCA is used as a dimensionality reduction technique for reducing the hyperspectral image dimension. The first principal component band, describing the maximum data variance direction is embedded with the segmentation map in CLUS-BPT framework in Chapter 3. Apart from reducing the number of dimensions, PCA ideally suppresses present noise in the hyperspectral data.

PCA transformation [52] diagonalizes a sample covariance matrix C of the original data X (of size $x \times y \times z$ in HSI case), thus determining its eigenvalues and eigenvectors decomposition. The covariance matrix is the measure of the correlation between two or more variables. It indicates if the variables change together in the same direction (high correlated) or in opposite directions (low correlated). Covariance value is always calculated between two variables, hence, the covariance matrix for n-dimensional data is made up of the covariances between every two variables and can be calculated using:

$$C(x_i, x_j) = \frac{\sum_{i=1}^{n} x_i x_j}{n-1}. \tag{A.1}$$

Eigenvalue decomposition of the data covariance matrix can be determined by using an eigenvalue and eigenvector decomposition method like Jacobi method, then the matrix C can be expressed as:

$$C = V \Lambda V^T, \tag{A.2}$$

where V is the eigenvector matrix and Λ is the matrix containing eigenvalues on the diagonal.
After computing the eigenvectors and ordering them by their eigenvalues in descending order, choose whether to keep all these principal components (eigenvectors) or discard those of lesser significance (of low eigenvalues). In this work, the number of PCA components used is 1. Hence, the highest data variance vector.
Finally, PCA transformation is:

$$X_{reduced} = V^T X, \tag{A.3}$$

where $X_{reduced}$ is a 2 dimensional image of size $x \times y$.

Appendix B

Gauss Jordan Elimination

Gauss-Jordan Elimination is an algorithm that can be used to solve systems of linear equations and to find the inverse of any invertible matrix. In this thesis, Gauss-Jordan elimination is used by Zynq PS to invert matrix A to be able to form the Laplacian matrix.

The idea here is to perform row reduction on the matrix $[A|I]$ where A is augmented with the identity matrix to obtain something of the form $[I|B]$. Then, $B = A^{-1}B = A$ provides a solution to the inverse problem. Row operations include a sequence of two steps:

1. Multiply one of the rows by a nonzero scalar.
2. Add or subtract the scalar multiple of one row to another row.

For example, consider $[A|I]$:

$$A = \left[\begin{array}{ccc|ccc} 2 & 6 & -2 & 1 & 0 & 0 \\ 1 & 6 & -4 & 0 & 1 & 0 \\ -1 & 4 & 9 & 0 & 0 & 1 \end{array}\right],$$

then, the first row operation is to make sure that the diagonal entry $A[0][0]$ is 1. This can be done by dividing the first row by 2.

$$\left[\begin{array}{ccc|ccc} 2 & 6 & -2 & 1 & 0 & 0 \\ 1 & 6 & -4 & 0 & 1 & 0 \\ -1 & 4 & 9 & 0 & 0 & 1 \end{array}\right] \Longrightarrow \left[\begin{array}{ccc|ccc} 1 & 3 & -1 & \frac{1}{2} & 0 & 0 \\ 1 & 6 & -4 & 0 & 1 & 0 \\ -1 & 4 & 9 & 0 & 0 & 1 \end{array}\right].$$

A[0][0] is called a pivot, and the elements under the pivot must be 0. This can be done by subtracting the first row from the second row. Furthermore, the first row can be added to the third row to obtain the necessary 0s in the first column:

$$\left[\begin{array}{ccc|ccc} 1 & 3 & -1 & \frac{1}{2} & 0 & 0 \\ 1 & 6 & -4 & 0 & 1 & 0 \\ -1 & 4 & 9 & 0 & 0 & 1 \end{array}\right] \Longrightarrow \left[\begin{array}{ccc|ccc} 1 & 3 & -1 & \frac{1}{2} & 0 & 0 \\ 0 & 3 & -3 & -\frac{1}{2} & 1 & 0 \\ 0 & 7 & 8 & \frac{1}{2} & 0 & 1 \end{array}\right].$$

Repeat the same process for the diagonal A[1][1] as a pivot. Divide row 2 (R_2) by 3, and compute ($R_1 - 3R_2$) and ($R_3 - 7R_2$).

$$\left[\begin{array}{ccc|ccc} 1 & 3 & -1 & \frac{1}{2} & 0 & 0 \\ 0 & 1 & -1 & -\frac{1}{6} & \frac{1}{3} & 0 \\ 0 & 7 & 8 & \frac{1}{2} & 0 & 1 \end{array}\right] \Longrightarrow \left[\begin{array}{ccc|ccc} 1 & 0 & 2 & 1 & -\frac{1}{3} & 0 \\ 0 & 1 & -1 & -\frac{1}{6} & \frac{1}{3} & 0 \\ 0 & 0 & 15 & \frac{5}{3} & -\frac{7}{3} & 1 \end{array}\right].$$

For diagonal A[2][2], divide row 3 by 15, and compute $(R_1 - 2R_3)$ and $(R_2 + R_3)$.

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 2 & 1 & -\frac{1}{3} & 0 \\ 0 & 1 & -1 & -\frac{1}{6} & \frac{1}{3} & 0 \\ 0 & 0 & 1 & \frac{1}{9} & -\frac{7}{45} & \frac{1}{15} \end{array}\right] \Longrightarrow \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{7}{9} & -\frac{1}{45} & -\frac{2}{15} \\ 0 & 1 & 0 & -\frac{1}{18} & \frac{8}{45} & \frac{1}{15} \\ 0 & 0 & 1 & \frac{1}{9} & -\frac{7}{45} & \frac{1}{15} \end{array}\right].$$

Thus, A^{-1} is:

$$\begin{bmatrix} \frac{7}{9} & -\frac{1}{45} & -\frac{2}{15} \\ -\frac{1}{18} & \frac{8}{45} & \frac{1}{15} \\ \frac{1}{9} & -\frac{7}{45} & \frac{1}{15} \end{bmatrix}.$$

The code listing below performs the Gauss Jordan elimination to obtain the inverse of a matrix A. The code is obtained from [48].

Code listing B.1: Gauss Jordan implemented in C.

```c
void GaussJordan (int size, double A[size][size], double I[size][size])
{
   int i, j, k;
double t;
for (i = 0; i < size; i++)
{
    for (j = 0; j < size; j++)  B[i][j] = 0;
    B[i][i] = 1; // forming identity matrix
        t = R[i][i];
        for(k = 0; k < size; k++) // diagonal matrix
                {
                        B[i][k] = B[i][k] / t;
                        R[i][k] = R[i][k] / t;
                }
        for(j = 0; j < size; j++)
        {
        if(i != j)
        {
                t = R[j][i];
                for (k = 0; k < size; k++)
                {  //
                B[j][k] = B[j][k] - B[i][k] * t;
                R[j][k] = R[j][k] - R[i][k] * t;
                }
        }
        }
}
}
```

Appendix C

Using HW/SW Co-design Implementation on Zynq Platform

This tutorial describes the setup steps for HW/SW codesign implementation of spectral clustering on ZedBoard development platform. The tutorial includes project creation, block design generation, simulation, synthesis and usage of Xilinx SDK Environment for interaction between software and hardware modules. In this tutorial, Xilinx Vivado v2019.1 (64-bit) is used.

C.1 Create Project

Open Vivado Design Suite and click on Create Project, a new project Wizard will pop up. Click next and select the Project location as the folder downloaded from GitHub repository available at [50]. The *VHDL* folder contains all necessary VHDL files, VHDL testbenches, and Tcl scripts required for Vivado Design Suite project setup. Click next and check on "Do not specify sources at this time" under "RTL Project" and then click next. Choose the Boards tab and locate "Zedboard Zynq Evaluation and Development Kit". Using the Tcl Console, run *project_SpectralClustering.tcl* script by typing *source project_SpectralClustering.tcl* on the console. This will recreate the project. The project is set to **xc7z020clg484-1** part number corresponding to Zedboard Zynq Evaluation and Development Kit board. First, the script will include all source files, IP repositories, default timing constraints, and simulation files. The Tcl script will also set *design_1_wrapper* as top module.

C.2 Synthesis and Simulation

Under ***IP Integrator***, click on *Open Block Design* to view the synthesized block diagram of the whole project design. The provided Tcl scripts create instances, set properties, and create interface and the necessary port connections between corresponding IPs within that block diagram. Figure C.1 shows the generated HW/SW co-design synthesized block diagram. Simulation files are provided for in the *Simulation Sources* folder of the Project. Chooses one of the sources as top-level for simulation by *Settings* $\longrightarrow$ *Simulation* $\longrightarrow$ *Simulation top module name*. An example waveform generated is shown in Figure C.2.

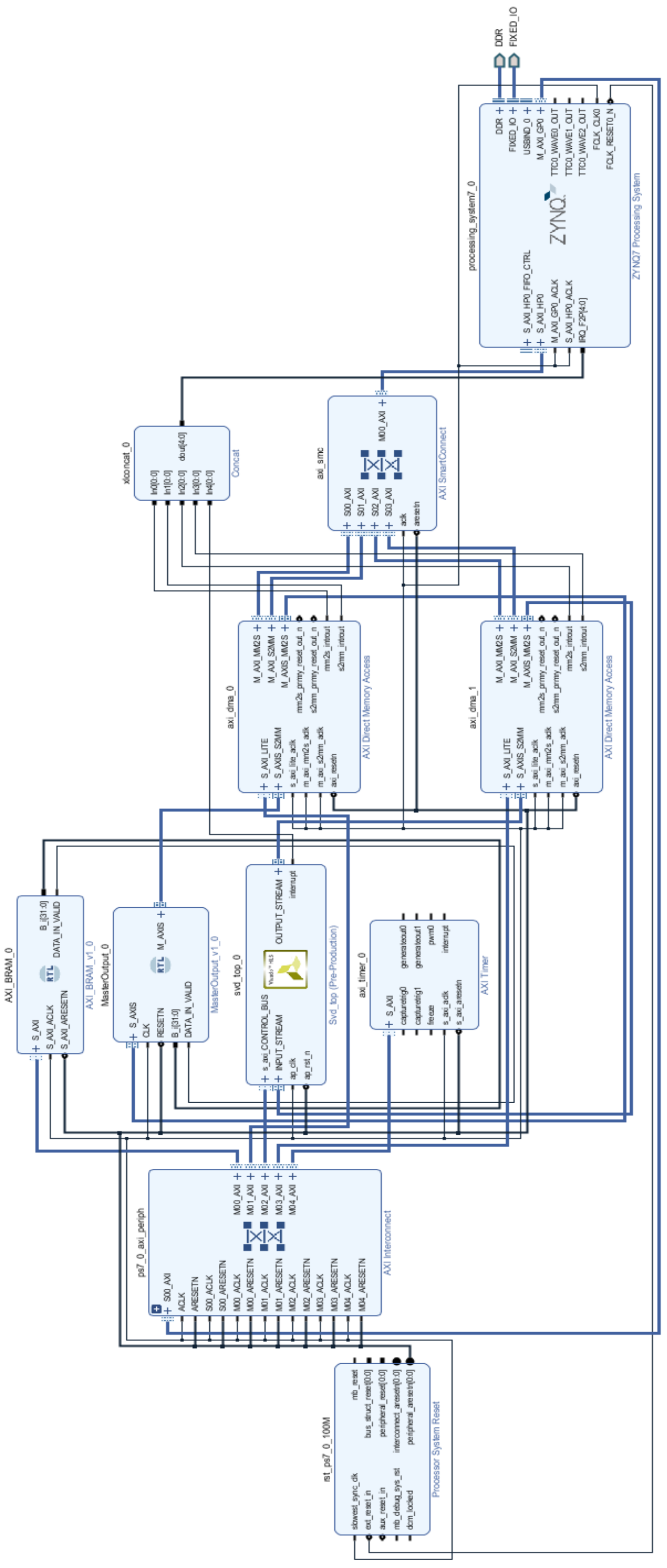
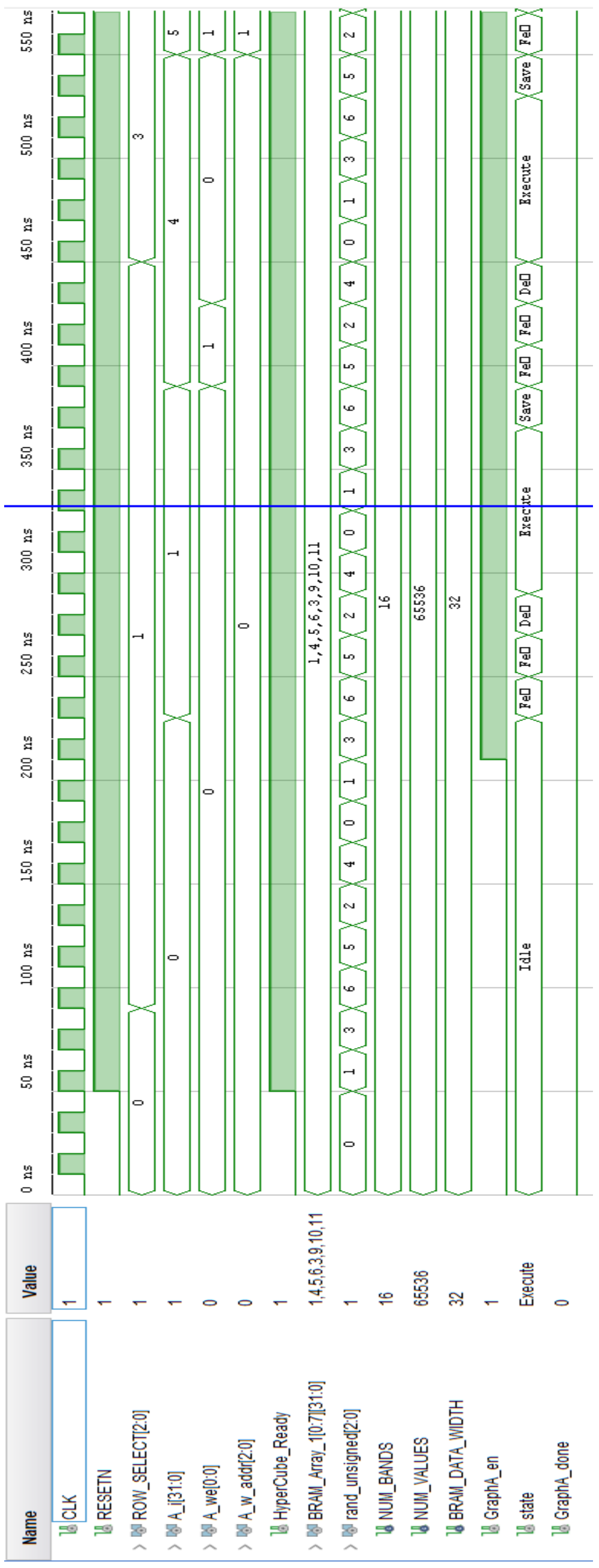


Figure C.1: Synthesis block diagram of HW/SW solution.

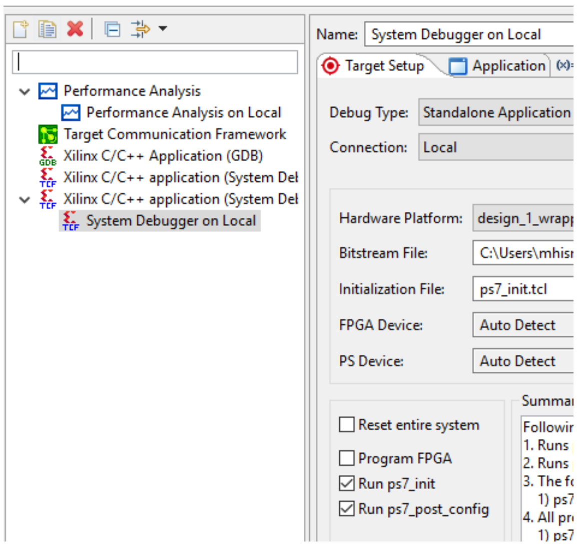


Name	Value
CLK	1
RESETN	1
ROW_SELECT[2:0]	1
A_i[31:0]	1
A_we[0:0]	0
A_w_addr[2:0]	0
HyperCube_Ready	1
BRAM_Array_1[0:7][31:0]	1,4,5,6,3,9,10,11
rand_unsigned[2:0]	1
NUM_BANDS	16
NUM_VALUES	65536
BRAM_DATA_WIDTH	32
GraphA_en	1
state	Execute
GraphA_done	0

Figure C.2: Graph construction simulation waveform

C.3 Post-synthesis and Implementation

Using Vivado GUI, run synthesis, implementation and generate bitstream. Upon synthesizing the design, post-synthesis resource utilization and timing reports can be viewed from *Open Synthesized Design*. Upon generating bitstream, the hardware is exported to be used by the Xilinx SDK. This is done as follows: *File* $\longrightarrow$ *Export* $\longrightarrow$ *Export Hardware*, make sure there is check on *Include bitstream*. Next, click on *File* $\longrightarrow$ Launch SDK.



- Performance Analysis
  - Performance Analysis on Local
- Target Communication Framework
- Xilinx C/C++ Application (GDB)
- Xilinx C/C++ application (System Del
- Xilinx C/C++ application (System Del
  - System Debugger on Local

Name: System Debugger on Local

Target Setup | Application

Debug Type: Standalone Application

Connection: Local

Hardware Platform: design_1_wrapp

Bitstream File: C:\Users\mhisr

Initialization File: ps7_init.tcl

FPGA Device: Auto Detect

PS Device: Auto Detect

☐ Reset entire system

☐ Program FPGA

☑ Run ps7_init

☑ Run ps7_post_config

Summar

Followir
1. Runs
2. Runs
3. The fc
1) ps7
4. All pr
1) ps7

Figure C.3: SDK - change project settings.

C.4 Xilinx SDK

The hardware platform created in Vivado can be found on the *Project Explorer* of Xilinx SDK GUI. Create a new application project by navigating to *File* $\longrightarrow$ *New* $\longrightarrow$ *Application Project*. A create project Wizard will pop up. Set the desired project name, OS platform to standalone and create new Board Support Package (BSP). Make sure that *design_1_wrapper_hw_platform_0* is selected as *Hardware Platform*. The *Language* is C. Click Next and choose an *Empty Application* template.

The application project is now created, but we need to modify the Board Support Package (BSP) settings to add support for SD card reading and writing. Navigate to *Project Explorer* tab and right click *<application project name>_bsp*. Click on BSP Settings. On the Overiview tab, check ***xilffs*** (Generic FAT file system library) and click OK. This will regenerate the necessary C libraries and drivers that will form the lowest layer of your application software stack.

Navigate to C/C++ Build Settings by right clicking on <application project name>. Under *ARM v7 gcc linker* $\longrightarrow$ Libraries, add ***m*** by pressing the add button. Press add again and add ***xilffs***. This will provide libraries for both math functions and SD card reading and writing functions. Under ARM v7 gcc compiler $\longrightarrow$ Optimization, set the optimization level to -O3.

Next, navigate to *Run* $\longrightarrow$ *Run Configurations* and double click on *Xilinx C/C++ application (System Debugger)*. System Debugger settings will be pop up as in Figure C.3. Make sure that *Reset entire system* and *Program FPGA* are unchecked as in the Figure. Navigate to the *Application* tab and click check on *ps7_cortexa9_0* processor. This will enable a user to compile any further edited C code (for exmaple, edit user defined parameters) before execution on Zedboard. Press close.

Copy the C files from the Github repository [50] to <application project name> on the Project Explorer. main.c file runs the program which reads the data from SD card, pre-processes the data, and initiates transfers using two different AXI DMA cores. The design is ready to run. Click on *Program FPGA* to upload the bitstream on the FPGA device and then navigate to *Run* $\longrightarrow$ *System Debugger on Local* and click Run.
After all the execution of k-means clustering, the program reports the execution time and writes the results to a file on SD card. The results can then be plotted on Python.