King Fahd University of Petroleum and Minerals

College of Computer Sciences and Engineering
Computer Engineering Department

COE 485: Senior Design Project
Final Report

FPGA-Based Clustering Machine

Semester 202

Group Members

Ahmed Alharbi	201636500
Hamza Alsharif	201642320

Supervisor

Dr Muhammad Elrabaa

Date: 10/05/2021

By signing the cover page of this report, you affirm that you fully agree with the content of this report, and that indeed it documents your own work.

Contents

1 Introduction

Recent developments in the field of ubiquitous computing and internet of things devices have introduced more methods in which data can be collected and stored. The result is that there are larger magnitudes of different types of collected data, and the storage capacities within data centers are growing tremendously. This has created new avenues for data exploration and increased the possibilities for finding useful statistical relationships using techniques from big data analytics, and machine learning. For machine learning applications in which the input data is stored in large data centers, the computational load required for training accurate high-end models is often spectacularly large. Such loads pose a problem for power consumption and similarly CO_2 emissions. A recent study on energy considerations in deep learning has shown that training a common NLP model using GPUs can emit 626,155 lbs of CO_2, which is the equivalent average emission of five cars throughout their entire lifetimes [12]. One reason for of these tremendous power loads and CO_2 emissions is that the computers used to train such models are general purpose machines that are designed to handle a variety of different tasks.

In our project, we propose a special-purpose computing core that performs a specific unsupervised machine learning task, namely clustering. One of the most popular methods for this task is to use the K-Means Clustering [8] algorithm. K-Means Clustering is an algorithm that takes an input dataset of multidimensional points and groups these points and classifies them together into k clusters. Clustering is widely used in many applications that involve finding statistical relationships or correlations between data points and grouping them into suitable categories which are represented by the clusters.

With the abundance of large corporate data centers there becomes a need to efficiently perform clustering on large amounts of data so that useful relational information can be extracted, while maintaining reasonable power consumption and without sacrificing speed. In this work, we propose an efficient clustering approach that utilizes an FPGA to work as a custom computing machine that can be attached to networks. We do this by making computational improvements on the original k-means algorithm to work better on hardware, and we introduce this design as a configurable IP core.

1.1 K-Means Clustering - The Algorithm

The K-Means Clustering algorithm involves grouping a given dataset of multi-dimensional points into a user-selected k number of clusters by computing the means or centroids for these clusters iteratively until they stabilize. A point is grouped into a cluster by finding and assigning the nearest centroid to it. Algorithm 1 below shows the steps involved, assuming three-dimensional data points within the range of 0 to 255. First k centroid values are initialized. Next the centroids accumulate the data points by determining the nearest centroid to each point using a distance metric calculation (euclidean, manhattan, etc.). After all the points have been accumulated, the centroid values (or means) are calculated by dividing the point accumulators by the number of points, and this is done for all of the clusters. This is done iteratively until all of the cluster's centroids converge, which for a single cluster occurs when the difference between the new centroid value and the previous iteration's centroid value is within a user-defined threshold. The clusters are then returned so that the points can be classified to each of the clusters accordingly.

Algorithm 1: K-Means Clustering

```
// assuming input data is stored in data variable
k = userInput()
threshold = userInput()
// threshold is the stability margin for centroid convergence
clusters = []
// clusters will contain centroid values, accumulators in x,y, and z, the
   number of points assigned to them, and the centroid values from the
   previous iteration
for $i = 0 \to k$ do
    clusters[i].centroid.append([randomInteger(0, 255),
    randomInteger(0, 255),
    randomInteger(0, 255)])
    clusters[i].previousCentroid.append([randomInteger(0, 255),
    randomInteger(0, 255),
    randomInteger(0, 255)])
    // initial centroid values are generated randomly here but in practise
       the initialization method can make convergence faster as well as
       ensure a good quality of the final converged solution
end
allStable = 0
while not allStable do
    foreach $point \in data$ do
        nearestCluster = findNearest(point, clusters)
        // can use any distance metric to find nearestCluster, such as
           manhattan, euclidean, etc.
        nearestCluster.accX += point[0]                    // point's x value
        nearestCluster.accY += point[1]                    // point's y value
        nearestCluster.accZ += point[2]                    // point's z value
        nearestCluster.points += 1
    end

    foreach $cluster \in clusters$ do
        cluster.centroid = [cluster.accX / cluster.points, cluster.accY / cluster.points,
         cluster.accZ / cluster.points]
        if $|cluster.centroid - cluster.previousCentroid| \leq threshold$ then
            cluster.stable = 1
            cluster.previousCentroid = cluster.centroid
        end
    end
    allStable = 1
    foreach $cluster \in clusters$ do
        if not cluster.stable then
            allStable = 0
            break
        end
    end
end
return clusters
```

1.2 Clustering in Hardware - Problem Scope

This project deals with the hardware implementation of K-Means Clustering. The main problem is that current algorithms that exist are sufficient but cannot easily be implemented into efficient high speed hardware circuits. This is in regards to speed, power, and resource utilization. Some applications are rather fast in their processing but utilize a large percentage of a device's hardware resources. Other implementations could have a small percentage of hardware utilization but may not be optimal in terms of their processing speed. Power is also a major issue here, as many implementations do not take into account the amount of lost power due to unnecessary bit-switching.

1.3 Similar Works in the Literature

The scope of clustering approaches has solid foundations within the literature, as others have proposed their own methods and design iterations. Some of these approaches have introduced complete hardware implementations, while other ones only suggest algorithmic improvements to the k-means clustering algorithm that are based on theoretical work. Work done by the latter group has most commonly been verified through software.

One hardware-based approach is the design proposed by Hussain et al. which follows a stage-based computation architecture [5]. These consist of the distance finder, minimum distance finder, accumulation, and divider stages. An input point arrives through a stream and the Manhattan distances [3] between it and each of the clusters is calculated. The minimum of these distances is then found through a hierarchical comparison scheme. The cluster associated with the minimum distance is then chosen to accumulate the point for the next iteration. After the arrival of the final input point the new cluster values are computed from the associated cluster accumulators and counters. The advantage of this design is that it can easily be pipelined, as well reconfigured to follow a parallel architecture. It has two major drawbacks however, in that it requires the calculation of k distances for each point, and that it also uses an integer division core which is highly inefficient. An algorithmic improvement that can be made to this design is to avoid computing distances as well as to perform as few comparisons between points and clusters as possible.

One way of making such an improvement is introduced by Saegusa and Maruyama in their kd-tree hardware implementation [11] based on the methods presented by Kanungo et al. [7]. The idea is to build a binary search tree with k dimensions, alternating these dimensions with level growth down the tree. The purpose is to perform a minimal traversal of this tree, visiting only the candidate set of clusters which can be considered to be neighbours to the input point. Through this technique, the number of distance metric computations is reduced since comparisons are done between a point and the candidate set cluster values instead of all possible existing clusters. We use these proposed methods as the basis for our approach and further improve on them by introducing pipelining and parallel computations. We also use a modified comparison scheme based on building a kd-tree, and finally we also eliminate division operations by avoiding the calculation of a mean value from cluster accumulators and counters.

1.4 Possible Approaches to Platform Selection

For the general problem of clustering, the most prominent approaches are to use a software application that runs on a CPU, or one that runs on a GPU, or a complete hardware-based approach using an FPGA. Software based approaches are usually sufficient for single-use scenarios, or for relatively small data sizes but suffer greatly when the size and quantity of the data increases. GPU-based applications perform much better due to their parallel threaded nature but are sub-optimal when it comes to power, since all of the GPU's components are always running simultaneously, even the resources that are not needed at a certain instance during processing. FPGAs on the other hand are very fast while also retaining power-efficiency. This is because they are designed to only utilize as much hardware as is required by a specific synthesized circuit, in addition to allowing for different runtime options.

FPGAs are also advantageous both for development and prototyping because they grant us flexibility in algorithmic development with the possibility of testing and comparing multiple designs in their throughput, power efficiency, and resource utilization. One major drawback to using FPGAs is that they are inflexible in resource allocation because once a circuit is synthesized, it has a fixed amount of resources allocated to it that cannot change. In our case, this is not an issue since we require our application to process large amounts of input data so we need to account for the worst case scenario by reserving a large percentage of the FPGA's resources. Table 1 outlines a summarised comparison between the possible platform approaches.

With that being said, we find that a tailored hardware implementation is the most effective choice for the design we are implementing. In order for the clustering circuit to be used conveniently and for a wide range of applications, there is also a significant benefit in designing it to be network attached. This would allow the input data to arrive to the circuit in a stream through the network, so that the clustered output data can be processed and transmitted with a speed that matches that of the network, therefore avoiding bottlenecks. To allow the user to conveniently access our hardware application, an API and web interface can also be developed. This provides an access method that lets users use it as a server over a network. In this initial phase, we are using image data as the input so that we can visualise the clustering result to conveniently identify any faults that may have occurred.

Approach	**Description**	**Pros**	**Cons**
Software (CPU)	Running a clustering software program to run on the CPU, written using a high-level programming language.	• Writing the software program will be a lot simpler. • Can use recursion and object-oriented programming. • Program development is flexible - can make changes very easily. • Portability is viable. • No need to purchase special equipment.	• Very slow and inefficient compared to GPU and FPGA implementations. • Cannot optimize processing for a specific algorithm. • Very limited parallel processing capabilities.
Software (GPU)	Running a parallel multi-threaded clustering software program to run on the GPU, written using a high-level programming language.	• High speed and throughput due to fully parallel architecture. • Ease of writing programs that can use multiple threads if independent operations are involved. • Flexible program development. • Can use object-oriented programming.	• High power consumption in comparison to FPGAs. • Expensive in comparison to CPUs. • No advantage to using it when mostly dependant computations are involved. • Supports only one type of parallelism design paradigm, which is single instruction multiple data (SIMD).
Hardware (FPGA)	Running an optimized parallel pipelined hardware program to run on an FPGA, written using a hardware description language (HDL).	• Fine-grained parallelism due to FPGA synthesizing as much hardware resources as required. • Can design very specific and optimized hardware circuits. • High speed and throughput with lower power than a GPU. • Can allow for independant specialised computing machine designs.	• Cannot allocate more resources during runtime (after circuit has already been synthesized). • Very expensive when compared to CPUs and GPUs. • Cannot be used for general-purpose processing.

Table 1: A comparison of the most common platform selections for applications. All of the mentioned points are applicable to clustering.

1.5 Potential Impacts

The main impact of this system is that it provides an efficient, convenient, and low-power solution for clustering. This is highly beneficial for data centers which have significantly large data stores and require high-speed real-time processing. Another impact is that this system can enhance research in artificial intelligence and machine learning applications, as it can process larger amounts of data with greater throughputs.

The system additionally has a broader societal impact mainly on the economy and the environment, as well as a possible social impact. The economical impact is significant as companies will benefit due to the increased capabilities of finding business insights from consumer or sales data. This will allow businesses to better tailor their products to consumers' needs, as well as uncover the business strategies that are most effective in generating revenues. Some common business use cases for clustering include recommendation systems, grouping customer demographics, grouping data from market research surveys, grouping products of similar types, etc.

Moreover, the low-power aspect of our system is beneficial for the environment because of the amount of saved electricity from reduced special-purpose processing within data centers. Reduced power consequently results in reduced CO_2 emissions, therefore our system can contribute to dampening the effects of climate change by decreasing greenhouse gas emissions. Furthermore, less generated power results in less generated heat which in turn reduces cooling costs within data centers.

There is however, a potential negative health impact that is the result of social media companies getting better at identifying how to maintain user engagement on their platforms. They can use that knowledge, which was obtained from clustering their user data that is saved in corporate data centers, to increase screen time on their respective platforms to catalyse social media addiction. Social media addiction has been shown to cause mental health problems due to negatively affecting self-esteem [4]. This impact however, is most likely minor in scale compared to the others.

Finally, our system has no direct observable negative impacts in the areas of public safety and welfare, as well as privacy/information security. These impact factors in different areas are summarised in Table 2 below.

Economic Impact	Improved capabilities for finding statistical relationships and insights for businesses, which leads to more profits. Enhanced business intelligence will also allow companies to better tailor their products due to better understanding of their consumers. All of this leads to positive economic growth.
Health Impact	Social media companies using our system will get more insights into improving user engagement on their platforms. More engaged users on social media could become addicted. Social media addiction has been shown to cause some mental health problems that stem from reduced self-esteem [4].
Environmental Impact	Our system is power-efficient, therefore reduces the amount of consumed power for processing. For data centers that have large data stores that need to be processed by our system, the accrued power savings are quite significant. These power savings are highly impactful with regards to preserving the environment, due to less required power from generating stations. Consequently, less power consumption translates to less greenhouse gas emissions, which in turn slows down the effects of climate change. Additionally, less power corresponds to less dissipated heat which in turn reduces cooling requirements as well as costs.
Societal Impact (welfare and safety)	No direct observable societal impact, or in welfare and safety.

Table 2: An outline of the different impacts our system would have in different areas.

2 Requirements and Constraints

For the system that we have designed, we have synthesized it on the Cyclone V board so that consistent testing results can be obtained. Additionally, this board was seen to be the fastest using the free version of the design software that was available to us. Therefore the requirements, constraints, and specifications are outlined based on our board choice. In reality, the system can run on any FPGA board but we have constricted ourselves to this one due to circumstantial limitations. For future work we will be synthesizing on better, faster boards that are more representative of the hardware that the system will run on in a practical scenario. Note that the specifications are mentioned in the system design section.

2.1 Requirements

The requirements for this project are as follows:

- The system shall be capable of processing as many clusters as possible
- The system shall be capable of receiving and processing as many points as possible.
- The individual dimensional values of the points will be 8-bit unsigned integers.
- The system shall be in the form of a standalone custom computing machine for clustering where the number of points and number of clusters are user-inputs evaluated at runtime.

- The system shall be capable of becoming a network attached computing machine that is interfaced through ethernet.
- The system shall be able to match the speed of high speed ethernet.
- The system shall achieve a point misclustering percentage error of less than 5%.
- The system shall not utilize resources outside of the FPGA.

2.2 Constraints

- The system is constrained to not having a constant throughput due to the dynamic nature of the data and number of clusters.
- The system is constrained to using the Cyclone V board.
- The system is physically constrained to using a maximum storage capacity of 0.8575 MB based on available BRAMs.
- The hardware circuit is constrained to using 342 DSP blocks.

3 Work Plan

3.1 Timeline

Week 1 The first week was dedicated to determining the senior project problem and defining it. By the end of the week, a project proposal/action plan was written up and sent to Dr Elrabaa for his approval.

Week 2 This week was dedicated to defining the problem and researching it. Relevant research papers were read to better understand some of the solutions and approaces done by others in the field. The first version of the golden model was written using Python. At first, this model performed a standard K-Means Clustering procedure and did not include any optimizations. It only had the feature to cluster images. It included options to use either the Manhattan or Euclidean distance. The github repository was also made.

Week 3 The first optimized design iteration was attempted in the software version. This iteration involved treating the cluster dimensions separately. The sub-box segment initialization method (Figure 9), was also implemented. There was a major error within the code and the reason for it was identified. This problem however was not easily fixable and it showed that we had misinterpreted the implementation of this method. A meeting with Dr Elrabaa was held to discuss this. After better understanding the problem, we were able to debug it and fix the software version. We tested it extensively on images with different values of K and with different distance metrics. The silhouette coefficient (Equation 1) was chosen as a design metric and was implemented within the code. A histogram plot was added to the code as well to better visualise the classification accuracy for each point.

Week 4 A newer cleaner version of the code was written, also in Python, as the previous one was getting messy and difficult to track/modify with growth. This new version also included the feature to generate random data samples for clustering instead of images. It also included the feature to plot the points on a 3D plane for visualisation. An approximation that was intended for convenient avoidance of division was also implemented into the code. The results seemed promising. We also edited the code so that it avoids division and we were successful in doing so.

Week 5 A computation mistake was identified within the code and was quickly fixed. It revealed that the results that seemed promising beforehand were not up to our clustering quality requirements. We had to re-think our approach to the problem. We discussed what we could do and started implementing the second design iteration which was the point streaming and expansion strategy. This strategy however was proving to be very troublesome so we decided to abandon it. Instead we went back to a paper that we had reviewed, that discussed implementing a kd-tree in hardware [11].

Week 6 The paper's kd-tree hardware implementation was not optimal and in some ways not practically applicable to our case, so we needed to think more generally about kd-trees as data structures. We attempted to implement a kd-tree in our software version a few times in different ways but were always unsuccessful.

Week 7 We found an open-source implementation of a kd-tree online and we tested it out to better understand how it works. After fully understanding it we integrated it into our code but with division. This worked very well and we were very happy with the results. We then tested it extensively with different cases. The mid-term report was also assigned during this week.

Week 8 We faced a problem when attempting to remove division from the previous successful kd-tree implementation. This was due to the cluster value approximation that we were using before not translating well to the kd-tree implementation. We attempted to debug the problem or find a better way to implement the approximation but the results were unsatisfactory. We were not able to solve the problem during this week.

Week 9 This week was dedicated to writing up the midterm report for submission. We worked on the required illustrations which were the use-case, activity, and deployment diagrams. We completed the mid-term report as required and submitted it by the end of the week.

Week 10 At the beginning of this week we analyzed our attempt at removing division from the kd-tree in more detail. We realized that our previous approximation is no longer applicable, but instead we can find an actual value for our solution to the cluster values. We implemented the actual value approach and found it to be very successful. We also tested it with the silhouette coefficient metric and found accurate results. This was a monumental achievement within the project because it meant that we could now finally start working on the hardware implementation. There was however, one issue that we had to think about. Our kd-tree implementation has a step that involves sorting the clusters before each iteration. We realized that this might complicate the design in hardware so we did some testing to determine whether this step was necessary. The conclusion that we came to was that it was necessary because it reduced the number of iterations significantly. We also decided that in order to evaluate the performance of our upcoming hardware solution we would need base designs without the improvements that we propose, to have a reference to compare to. Therefore, we promptly began designing a basic sequential clustering hardware implementation.

Week 11 We completed the sequential design during this week and began working on another design with the improvement of only removing division. We completed the second design without division and started thinking about how to approach the kd-tree implementation in hardware. We did this by drawing the proposed architectural model. After completing our reference model diagram, we promptly began work on implementing it.

Week 12 This week was fully dedicated to the kd-tree hardware implementation. We began by designing the cluster_ce and cluster_pe modules. Our objective was to first ensure we had a way to test the sorting stage in hardware so we designed these modules to first meet these requirements. We finished their design and verification by the end of the week and began integrating them together into the node module.

Week 13 We continued work on the node module so that it would also accommodate point propagation. We completed a working version of the sorting and point propagation stages of it and we spent the remainder of the week testing it. The testing stage took a considerable amount of time because there were many scenarios that we had not thought of before so we had to account for them. This resulted in continuous revisions and iterations of our working design. We also spent a significant amount of time this week finishing up the final report so that we could have it ready for submission on time.

3.2 Task Division

The project tasks were completed by either Ahmad, or by Hamza, or were joint efforts by both members. While it is difficult to record every single contribution and to correctly attribute valid credit for all the tasks in a collaborative project such as this one, an attempt has been made at doing so. This is outlined as follows:

Ahmad's tasks

- Assisted in writing the action plan for the project.
- Supervised in writing the first design optimization that involved clustering separately on each dimension.
- Identified the reason for the major error in the first design optimization.
- Wrote the second cleaner version of the golden model.
- Wrote the code for the 3D plot for point and clustering output visualization.
- Wrote the code to generate random data samples and to control the standard deviation of their distribution.

- Edited the code in order to remove integer division
- Identified and fixed a computation mistake within the code related to using an incorrect method to verify cluster output quality, which allowed us to abandon the separate dimensional processing approach.
- Explored, implemented and tested the point streaming and cluster expansion approach.
- Identified the correct way to remove division from the kd-tree approach.
- Revised the mid-term report.
- Designed the sequential k-means clustering model with division.
- Designed the sequential k-means clustering model without division.
- Integrated the cluster_ce and cluster_pe modules together and used them to design the node module.
- Came up with an internal communication scheme within the tree between different nodes.
- Wrote a python code to generate the text to declare and synthesize multiple node modules to make a tree.

Hamza's tasks

- Wrote the action plan for the project.
- Wrote and tested the first version of the golden model.
- Wrote the first design optimization that involved clustering separately on each dimension.
- Fixed the major error in the first design optimization.
- Implemented the silhouette coefficient metric within the code.
- Added a histogram for visualization of silhouette coefficients for all of the points. Also computed the average silhouette coefficient.
- Added a method to identify misclassified points and to calculate their average.
- Implemented the approximation method for conveniently getting rid of integer division.
- Found and separately tested an open-source code for a kd-tree implementation. Decided to integrate it into our the software model code base.
- Attempted to remove division from the kd-tree method using the prior approximation.
- Wrote the mid-term report.
- Evaluated the necessity for sorting within the kd-tree and determined that it was necessary.
- Wrote a pseudo-code algorithm for our approach to using a kd-tree for k-means clustering in hardware.

- Drew the architectural model schematics for the kd-tree hardware design.
- Implemented the cluster_ce and cluster_pe hardware modules that make up the node module.
- Modified the manhattan module to fit our design accordingly.
- Wrote testbenches for the cluster_ce and cluster_pe modules to verify their operation.
- Supervised and directed the integration of the cluster_ce and cluster_pe modules into the node module.
- Wrote the final report.

Joint tasks

- Read research papers and reviewed the literature.
- Discussed fundamental design decisions of both the software and hardware models.
- Wrote, debugged, and tested large portions of both software versions, and the hardware designs.
- Attempted multiple iterations of failed kd-tree implementations.
- Integrated the open-source code into the software code base and tested the model after its integration.
- Designed the use-case, activity, and deployment diagrams.
- Removed division from the kd-tree approach in the correct fashion. This allowed for the transition to the hardware version.
- Discussed and formulated the schematics for the kd-tree hardware design.
- Decided to move the internal components of the cluster_ce out to become part of the node module instead of its own self-contained module.

4 System Design

4.1 System's Behaviour: Use-Cases

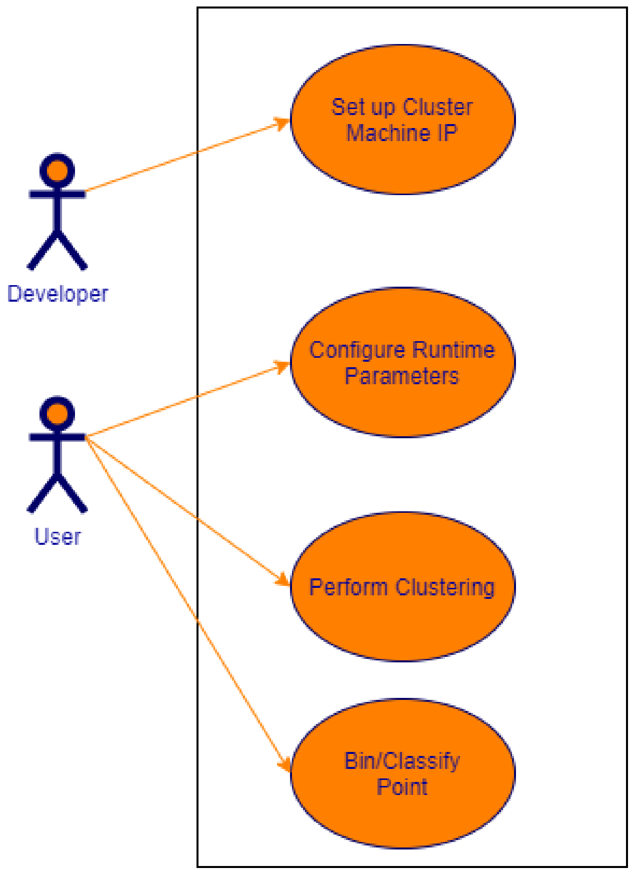


Figure 1: Use Case diagram for the system.

The use case diagram for our system consists of four primary use cases and they are shown in Figure 1. The first is to configure the cluster machine IP, and this is done by the system developer using the APIs provided by the cluster machine IP. Next, the user can configure the runtime parameters, which are the segments in each dimension. They can then run the clustering machine to train a cluster model on the FPGA and to receive the segmented data. Finally, if a pre-trained model already exists, the user can also perform binning or data classification by sending a point as an input to the cluster machine to receive the cluster that corresponds to it.[1]

[1]These are the typical use cases for a complete networked computing machine, which is how we intend our system to be used. In our case, we are only able to provide and demonstrate the last two use cases since we were not able to get access to an FPGA with which we can install our IP core. Therefore there was no practical way for us to set up the system as a cluster machine nor to configure its runtime parameters.

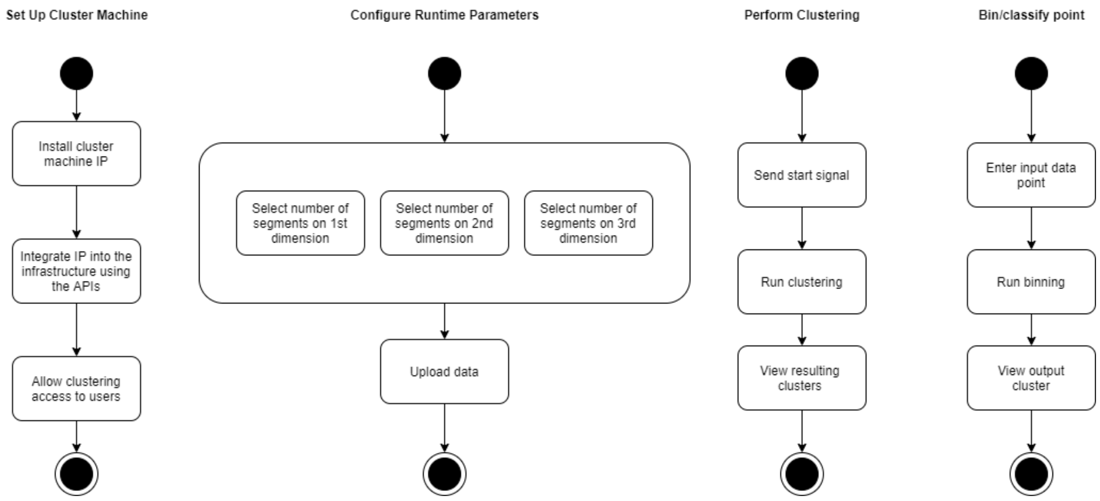

Set Up Cluster Machine
Install cluster machine IP
Integrate IP into the infrastructure using the APIs
Allow clustering access to users
Configure Runtime Parameters
Select number of segments on 1st dimension
Select number of segments on 2nd dimension
Select number of segments on 3rd dimension
Upload data
Perform Clustering
Send start signal
Run clustering
View resulting clusters
Bin/classify point
Enter input data point
Run binning
View output cluster

Figure 2: Activity diagram for each of the use-cases.

For each of the respective use cases in Figure 1, the activity diagrams are shown in Figure 2. Developers can set up the cluster machine by first installing its IP core, then by integrating it into the system using the provided APIs, and finally by providing clustering access to the users through utilization of the APIs.

Users can then configure the runtime parameters which in our case are the number of initialization segments in each dimension. In theory and with further development, the circuit can be generalised to support a variable number of dimensions but this will require additional work and is beyond the requirements that have been set for this project. After the runtime parameters have been configured users upload their data into the system.

When the cluster machine has been set up and the runtime parameters have been selected the user can initiate the clustering process, which will send the start signal. Once clustering is complete the user can view the final resulting clusters.

Once a cluster model exists on the FPGA, users can input new individual points to be classified by the pre-trained model. So in this case, a user will enter the input point, point classification will occur through binning, and the respective cluster that the point belongs to will be sent back as an output.

4.2 Product Concept and System Architecture

A typical system using the FPGA clustering IP core is comprised of mainly a web server, and the FPGAs running the cluster machine - this is shown in Figure 3. A user will first configure the runtime parameters and will upload the data as shown in the second use case in Figure 1. Clustering can then be initiated and performed as in the second use case. The input data, which is shown as an image in Figure 3, is received through the web interface which has already been configured by the programmer into the system in the first use case. The web interface has a pre-configured backend which interfaces with the server's web socket. This web socket is used to communicate with a router which manages incoming and outgoing loads between the numerous connected FPGAs. The router in turn communicates with the FPGAs through their ethernet interfaces. Inside an FPGA data received from the ethernet interface is saved onto memory, so that the K Means Cluster Processors have access to it. This will cause a limitation based on memory size within the FPGA, but in this case we propose clustering a portion of the dataset for training based on the maximum available memory. The cluster processors then perform the algorithm to generate the clustered model and centroid values accordingly. Once binning is required as shown in the fourth use case, the ethernet interface will communicate with the cluster processor directly, since the cluster processor contains the pre-trained clustered model. The cluster processor will send the classification output back directly through the ethernet interface.

One point to note is that the maximum possible value for K within a tree in hardware is a synthesis parameter, and can be specified based on availability of hardware resources on the FPGA board. The actual value for K that a user requires will be a runtime parameter that is specified by the user before the clustering task starts. This input value for K must be less than the maximum value for K specified during synthesis.

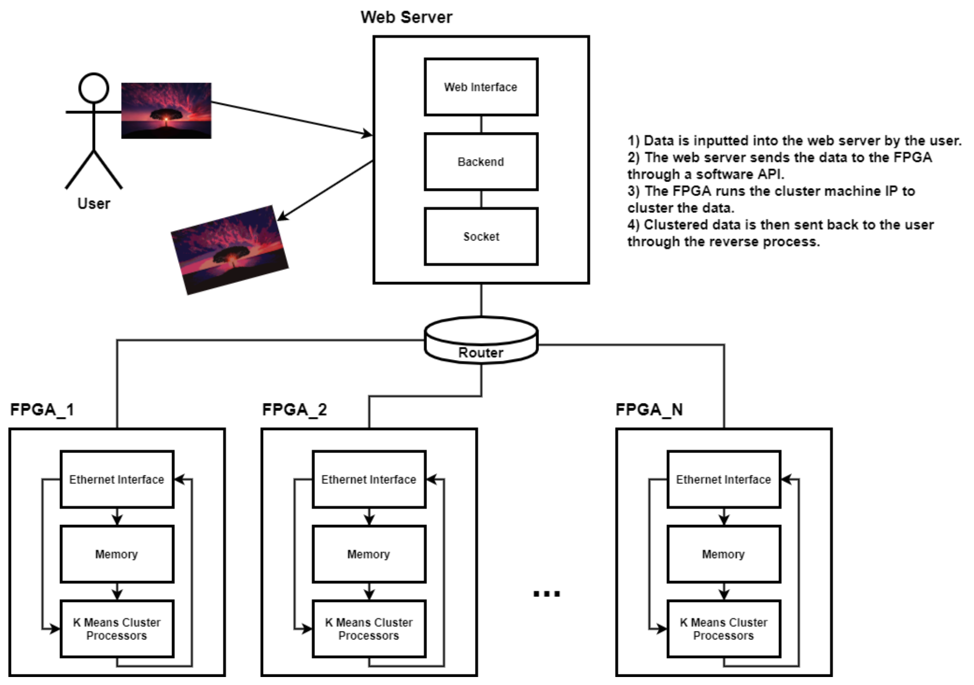

Web Server
Web Interface
Backend
Socket
User
1) Data is inputted into the web server by the user.
2) The web server sends the data to the FPGA through a software API.
3) The FPGA runs the cluster machine IP to cluster the data.
4) Clustered data is then sent back to the user through the reverse process.
Router
FPGA_1
FPGA_2
FPGA_N
Ethernet Interface
Memory
K Means Cluster Processors
...

Figure 3: Product concept to illustrate the system architecture.

4.3 System Specifications

- The system achieves a best case processing speed of 173 Mb/s within the constraints of the Cyclone V board.
- The system is capable of processing a maximum of 114 clusters based on the available resources on the Cyclone V board.
- The system is capable of storing and processing a maximum of 285833 data points based on

the space available in the BRAMs. BRAM management is performed by the synthesis tool.

- The input points for the system will be three-dimensional.

4.4 Compliance with Standards

The system hardware implementation is written using the 1364-2005 - IEEE Standard for Verilog Hardware Description Language. A typical implementation of the system IP uses 40 Gb/s ethernet compliant with the IEEE 802.3ba-2010 standard. It also uses the UDP protocol defined as RFC 768 by John Postel.

5 Hardware Algorithm Development

The algorithmic development stage was the longest in this project and was based on trial-and-error, experimentation, iterative design, as well as testing and verification. This is because we had requirements that we placed on the algorithm in order for it to achieve its intended functionality within the proposed system. Therefore, we needed to ensure that these algorithmic requirements were met before we could begin work on the hardware design. These algorithm requirements are outlined in the following subsection.

5.1 Guidelines and Design Decisions

Guidelines The general algorithmic guidelines are as follows:

- The generated cluster model for an input dataset must be of high-quality. In other words, points within a cluster should be close to that cluster's centroid and far from the other clusters' centroids. Point classification must also be accurate.
- The cluster algorithm's hardware implementation must not have a high power consumption.
- The cluster algorithm's hardware implementation must be fast. In other words, the algorithm must avoid lengthy, unnecessary, and inefficient operations.

Design Decisions Based on these requirements, we placed these following design decisions on the hardware algorithm:

- The algorithm must avoid integer division operations.
- The algorithm must avoid square root operations.
- The algorithm must avoid excessive multiplication operations - it should use as few as possible.
- The algorithm must avoid unnecessary bit-switching and should disable parts of the circuit that are not running at any given instance.
- When determining the nearest cluster to a point, the algorithm must avoid visiting clusters that are deemed to be highly unlikely results - due to being too far away based on the distance metric.

Reasoning behind the design decisions Integer division and square root operations in hardware are not optimized and are inefficient. An LPM_DIVIDE megafunction block takes up a significant portion of hardware resources, is high in latency, and for a pipelined version, takes more cycles to produce the final solution than we deem to be acceptable for our speed requirements [1]. The square root core is similar, and we avoid using it by computing the Manhattan distance instead of the Euclidean distance when searching for the nearest cluster.

Multiplication operations are optimized in hardware using multiplier DSP blocks so are suitable for replacing division operations, by implementing substitute arithmetic expressions based on multiplying both sides of an equality to get rid of the denominator. Too many substitute multiplications however can increase bit-width considerably, and can also unnecessarily increase the number of cycles. Therefore excessive multiplications should be avoided.

Based on transistor-level circuitry, power is dissipated when a bit switches from low to high or vice-versa. Therefore we aimed to reduce power consumption by disabling inactive parts of the circuit at any given time, thereby reducing unnecessary bit-switching. Additionally, by choosing to filter out as many far away clusters as possible during nearest cluster determination, we effectively reduce the number of computations and consequently the number of cycles, as well as the number of unnecessary bit switches.

5.2 Experimental Software Version - Golden Model

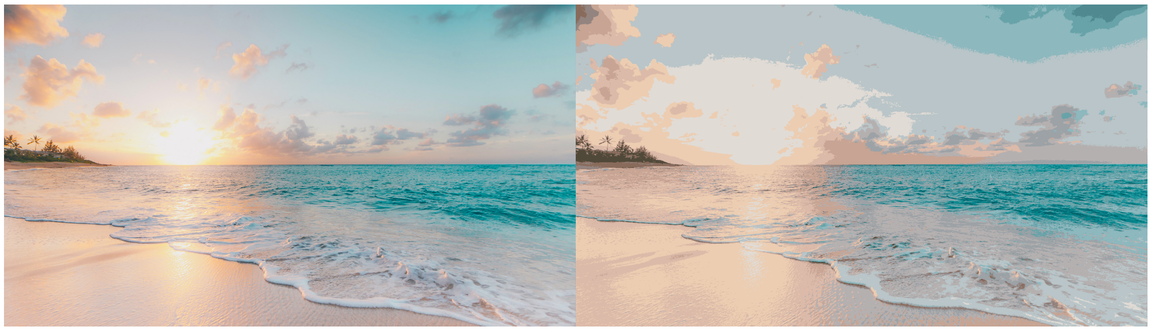


Figure 4: A side-by-side comparison of an input image and it's segmented output obtained from the first golden model that did not yet include any code optimizations for hardware. The value of K for the segmented image is 16.

To test out different clustering approaches and to have a baseline model with which we could refer to, a software version was developed in Python to act as our golden model. Algorithm iteration, experimentation, and verification were done using this golden model. Our first version of golden model implemented clustering in its most basic form and without any design improvements nor iterations. It had the feature to use images as an input for clustering, or in this case segmentation. A segmented image with its original input, from the first working version is shown in Figure 4.

As our code base began to grow we needed it to have a more general and organized structure that would allow for convenience in edits and changes. Therefore, we wrote a second modular object-oriented version of the golden model. We included some useful features for experimentation in this version. The first of these is the feature to generate sets of random normally distributed clusters for testing. This was done using standard functions provided by the scikit-learn library [9]. It was now possible to test two base scenarios, using either an input image or a randomly generated dataset. The second feature was to plot the output clustered data on a 3D plane for better visualization. The plotting library we used for this was plotly [6]. This 3D plane plotting representation is shown in Figure 5. The picture on the left shows the empty plot, and the picture on the right shows the plotted and clustered points when the input was an image. Additionally, Figure 6 shows the plotted and clustered points using a randomly generated dataset.

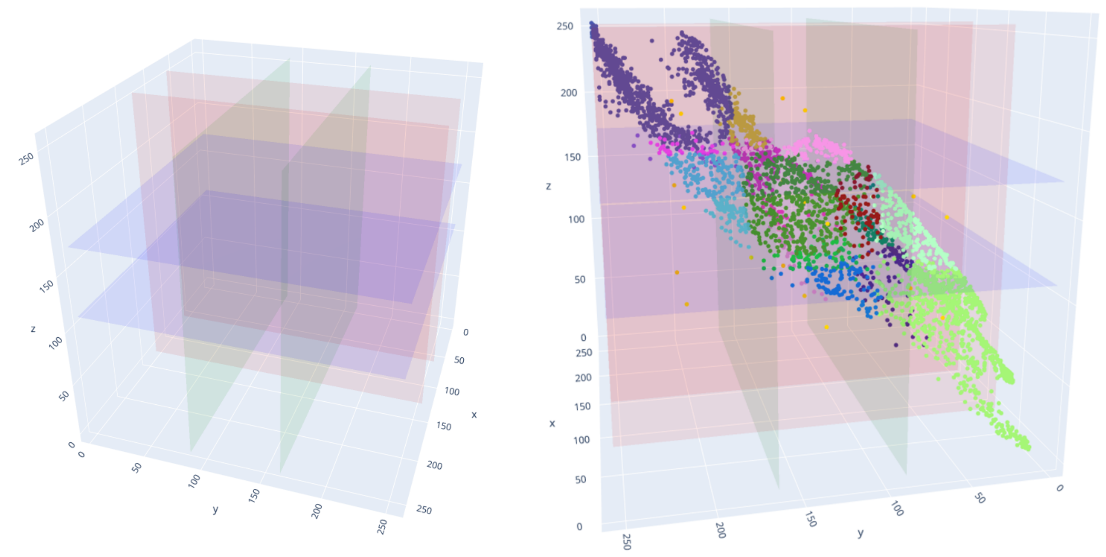


Figure 5: 3D space plot to visualise clustering outputs. The cluster example shown on the right is for an input image.

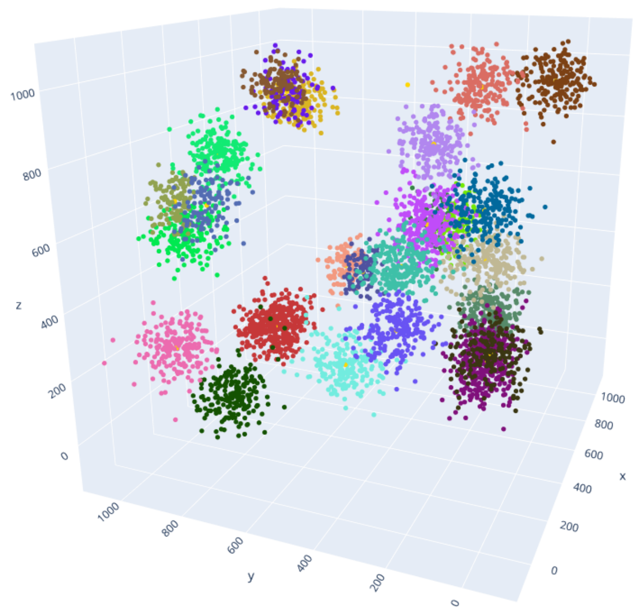


Figure 6: 3D Cluster plot with randomly generated data.

5.3 Clustering Quality Evaluation - Silhouette Coefficient

To evaluate the accuracy of the clustering output result, a quality metric was required. For clustering applications one of the most commonly used quality metrics is the silhouette coefficient [10]. The silhouette coefficient describes how compact and self contained clusters are, or in other words how close points within a cluster are to the cluster's centroid and how far away they are from other centroids. The equation for calculating the silhouette coefficient for a given point x is given as follows:

$$s(x) = \frac{b(x) - a(x)}{max\{a(x), b(x)\}} \tag{1}$$

Where $a(x)$ is the euclidean distance between x and the cluster it is assigned to, and $b(x)$ is the euclidean distance between x and the next closest cluster to it. $s(x)$ is in the range of $(-1, 1)$ and the more positive it is, the better the quality of the solution. A negative s value indicates a poor quality solution.

Next we needed a method to visualize the computed silhouette coefficients for all of the points, so we plotted a histogram with 60 bins and observed the frequency of different silhouette coefficient values. An illustration of the histogram plot is shown in Figure 7. Using the silhouette coefficient values computed from all of the points, we compute the average silhouette coefficient value. A remark to note is that if the points in the dataset are naturally close to each other then the average silhouette coefficient will not be very high, although this does not necessarily indicate a poor solution due to the points' natural close proximity. A high relative frequency of negative silhouette coefficient values however, is an indicator of a poor solution.

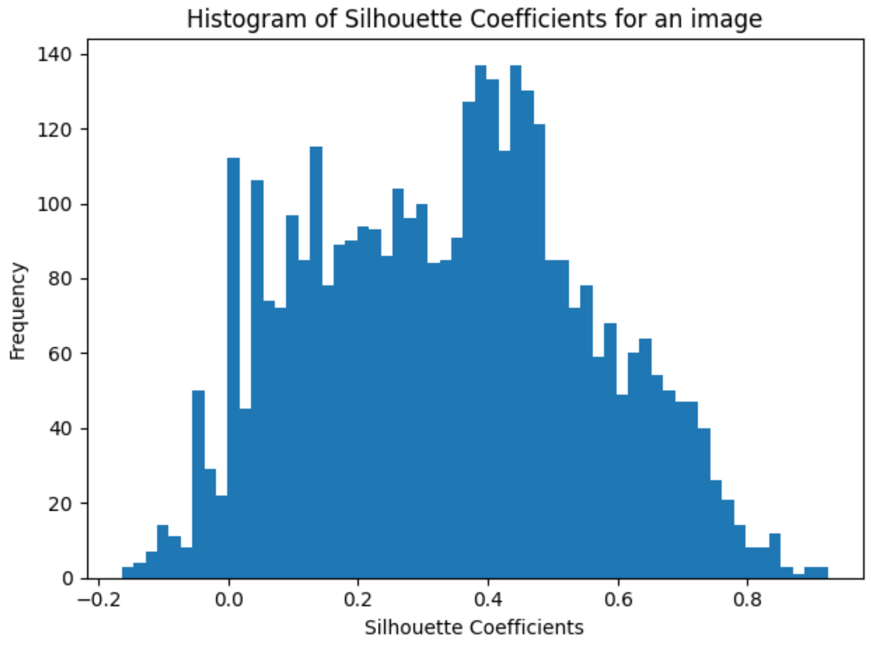


Figure 7: Histogram plot of silhouette coefficient values for all of the points in an early test instance.

5.4 Cluster Initialization

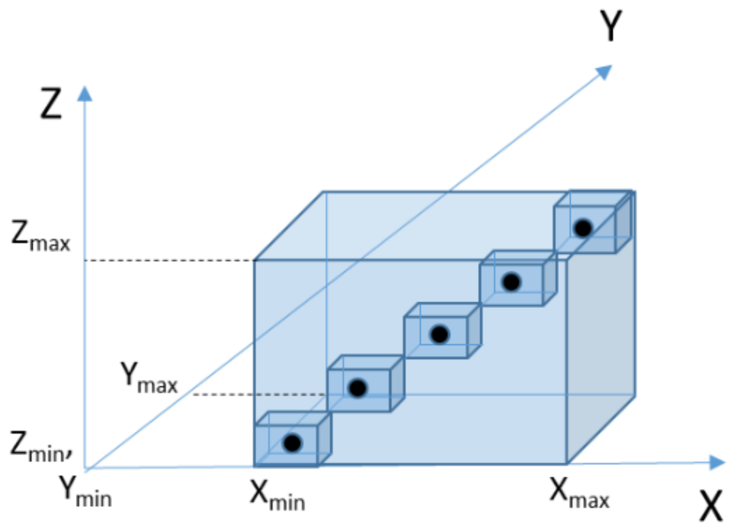


Figure 8: Diagonal initialization. Clusters are initialized along a diagonal line across the bounding box. Diagram credit - Dr Muhammad Elrabaa.

The initialization stage is critical for an optimal clustering strategy. An effective initialization can reduce the number of iterations significantly and can produce a higher quality solution. A poor initialization on the other hand, can increase the processing load excessively and can cause the cluster centroids to converge at sub-optimal positions. In our first version of the algorithm we were generating the initial clusters randomly and this is not a very good strategy.

An effective method at narrowing down the range with which clusters can converge in involves finding the maximum and minimum values for each dimension. These can then be used as boundaries to form a minimum sized cuboid, or a bounding box, that contains all of the points in the dataset. Using the bounding box as our operation space we initialized the clusters along equally spaced points between a diagonal line along the bounding box. This initialization scheme is shown in Figure 8. However, this method is not much of an improvement over the random initialization because it assumes that points will most likely be positioned along or close to the diagonal.

A better initialization strategy is to split the bounding box into smaller boxes and to treat these boxes as initial clusters. This is done by splitting each dimension into segments and finding the midpoints of these segments. The midpoints of each dimension are then combined to form the initial centers. This is illustrated in Figure 9. This strategy is generally better than diagonal initialization because it accounts for more evenly distributed points within the bounding box.

However, there exists a notable problem with this initialization scheme and it is that some of the initial clusters will not move from their place, and will be considered to be empty clusters. This problem is outlined in Figure 10. The black dots represent initial clusters that have not moved. The reason for this is the diagonal shape of the data distribution in 3D space. Since the initial clusters are outside of the data diagonal, they will never get any points assigned to them and will always be considered to have converged. This is against the requirements because in many cases the actual value for the number of clusters will be less than the user-defined value k. Therefore, to fix this problem there needs to be a way to determine the point distribution and ensure that the clusters are not initialized outside of it. For our senior project, solving this issue was beyond our scope of work, however we plan to tackle it in the project's future work as this is definitely an area of innovation.

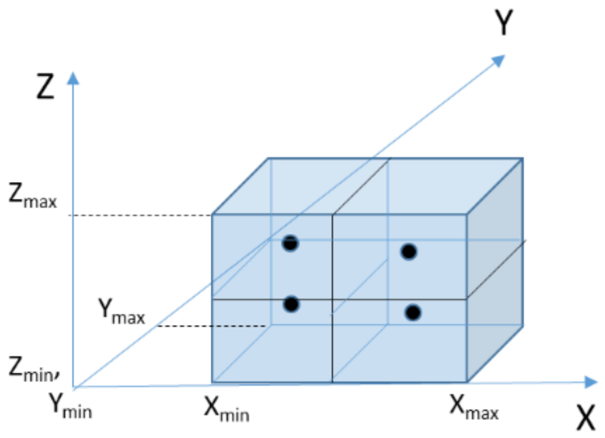


Figure 9: Sub-box initialization. Clusters are initialized by splitting the bounding box into smaller sub-boxes and using the centers of these sub-boxes as initial centers. Diagram credit - Dr Muhammad Elrabaa.

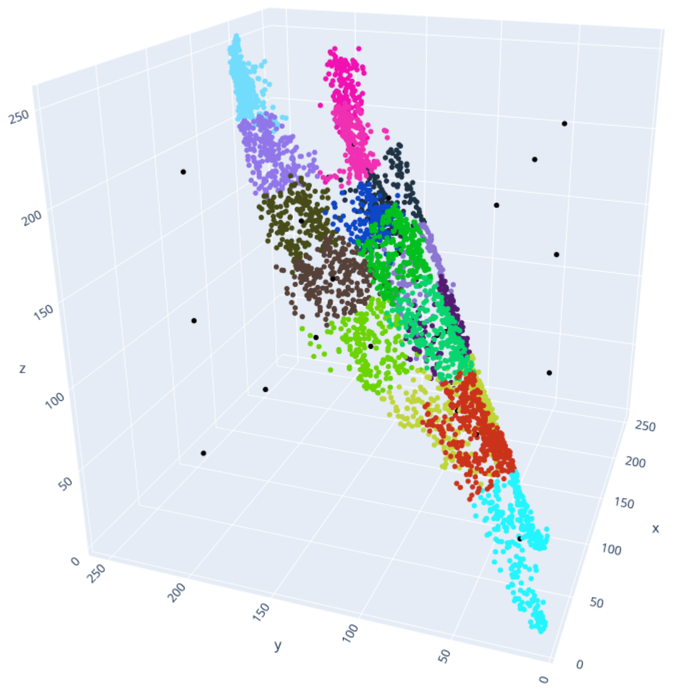


Figure 10: An example of when the box initialization method fails. The black dots represent the initial clusters that did not move from their place. This is because they are outside of the data range so no points will be assigned to them.

5.5 Avoiding Integer Division

One of our algorithm requirements as specified earlier was to remove integer division due to how taxing it is on hardware circuits. In the k-means algorithm, integer division is needed for computing the means from the accumulators and counters after each iteration. This was seen to be necessary because of the step in which we compare cluster centroids with points to determine which clusters the points should be assigned to. However by multiplying both sides of the comparison inequality, we can avoid the division operation and instead turn it into a multiplication. So as a demonstration, this is the original inequality:

$$P_d \leq \frac{S_d}{N} \tag{2}$$

Where P_d is the value of the point in dimension d, S_d is the accumulated value of a cluster in dimension d, and N is the number of points assigned to the cluster.

This inequality becomes:

$$P_d * N \leq S_d \tag{3}$$

In this scenario, we avoid integer division by turning it into multiplication which is much more efficient in hardware. The tradeoff here is that the bit-width for the value at the left of the inequality increases, in addition to the convergence condition now being determined from the cluster accumulators and counters. The system can then output the final converged accumulators and counters for the clusters, from which the means can be computed externally.

5.6 Clustering Design Iterations

Throughout the algorithm development phase, we were constantly making changes and experimenting with them to see their viability in becoming our final hardware implementation. Therefore we spent the most amount of time during the project's development on this phase, because some designs would fail and we would have to re-think our approach. From the beginning of the project up until its completion we attempted two primary design approaches, with only the second one satisfying our algorithm requirements. These algorithmic approaches were the dimension separation (Section 5.6.1), and clustering using a KD-Tree (Section 5.6.2).

5.6.1 Dimension Separation - Three Binary Trees

This method involves treating each dimension separately and performing three independent clustering procedures in parallel for each of them. Moreover, the initialization step is similar to the sub-box strategy of finding midpoints for sub-boxes with the difference being that these midpoints will be treated independently as initial centroids for their respective dimensions. In this case, we call them segments. The number of segments in each dimension is defined by the user. Clustering will then be done using these initial segments individually and in parallel for each dimension until the convergence condition is met for the three dimensions. The final clusters will then be built from all possible combinations of these converged segments. As a result, k will be equal to the product of the number of segments in each dimension as specified by the user.

In each dimension, clustering is done using a balanced binary search tree [2]. Binary search trees are useful because, assuming a balanced case, they reduce the worst case complexity for nearest cluster search from $O(n)$ to $O(log(n))$ which is excellent for performance. To build a single dimensional tree, we first calculate the midpoints between each two segments for all of the segments that we have. Next, we use these midpoints as comparison values for all nodes within the tree except the leaves. The leaves will instead represent the actual segment values. This is demonstrated in the displayed example tree in Figure 11. C_i is the cluster centroid value for cluster i in one dimension d. We assume that the array containing C_i values is sorted in ascending order. Given that the comparison values for each internal nodes are the midpoints between two clusters, then by directly comparing the respective dimensional value of the point P_d, we can determine whether P_d should go left or right down the tree. The leaf that P_d ends up in is the nearest segment to P_d. P_d is finally accumulated to that segment.

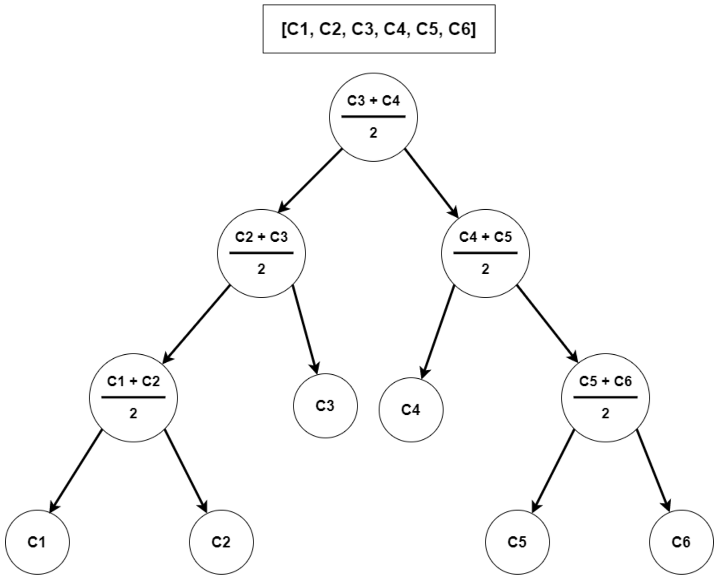


Figure 11: A representation of a midpoint-based binary tree used in the dimension seperation approach. If P_d is smaller than an internal node value, P_d will traverse to the left of that node. If it is greater, it will traverse right. The leaf that P_d ends up at is the segment it is assigned to, so it accumulates P_d.

Once all points have been assigned to clusters, the iteration ends and the convergence condition can be checked. Once all of the segments have converged, clusters are made from them using the combinations of all possible segments in each dimension. For example, for two segments $C1$, and $C2$ in each dimension the final combination of clusters would be:

$$\begin{gathered}[(C1_x, C1_y, C1_z), (C1_x, C1_y, C2_z),\\ (C1_x, C2_y, C1_z), (C1_x, C2_y, C2_z),\\ (C2_x, C1_y, C1_z), (C2_x, C1_y, C2_z),\\ (C2_x, C2_y, C1_z), (C2_x, C2_y, C2_z)]\end{gathered}$$

This is considered to be an approximation to the optimal solution, but when we tested it we had moderately acceptable silhouette coefficient values (no points were misclassified). Figure 12 shows the silhouette coefficient histogram using this method for randomly generated data.

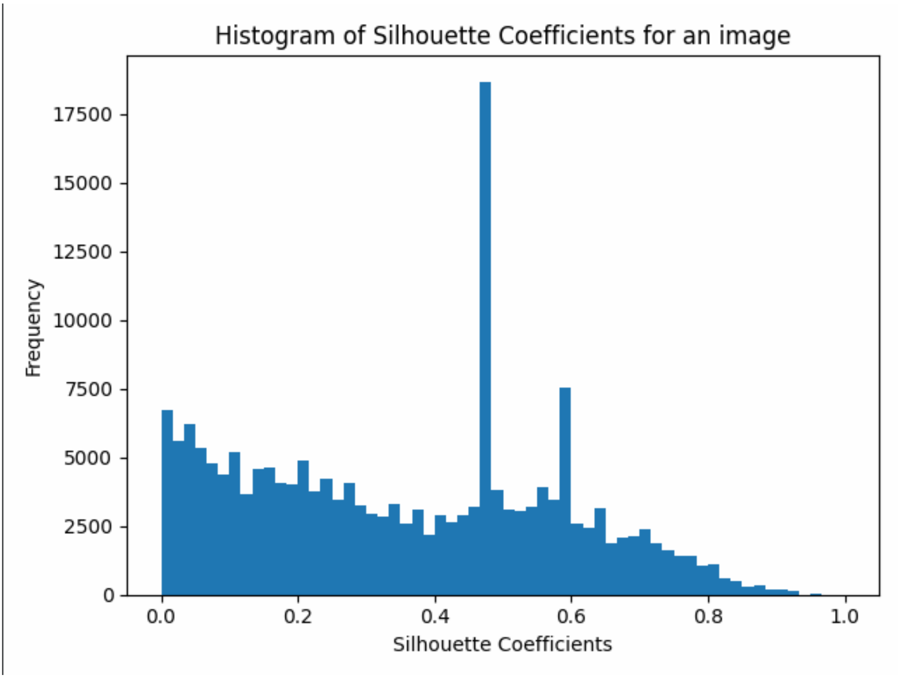


Figure 12: Histogram plot generated from random data and based on the dimension separation method. This instance was for 3 segments on the x, y, and z axes respectively. The average silhouette coefficient for all of the points is 0.355 . No points were misclassified.

One major disadvantage to using this method is that the final combined clusters can only have centroid values from their specified clustered segments in each of the dimensions. This limits the possibilities for the centroids and constrains them to only being a possible combination from the final segments. This issue can be seen visually in Figure 13. In the image, three segments are used for x, y, and z dimensions so K should be 27, but a limited colour range is seen in the image because many clusters will have the same values for a given dimension. This is prominent in the area at the center of the tree where the sun is situated. The orange hue of the sun was replaced with the red colour of the sky due to the limited output colour set.

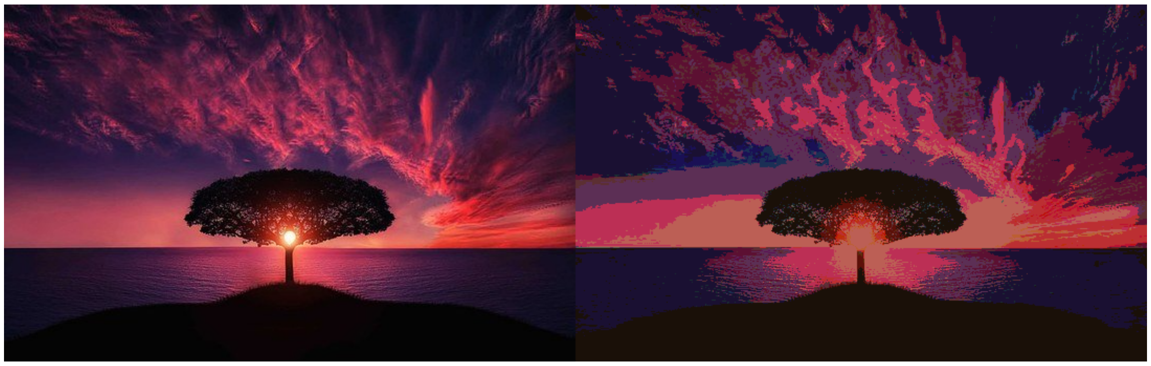


Figure 13: Side-by-side comparison of between an image and its clustered output using the dimension separation technique. This highlights the drawback with the the method, in that a limited value-set is generated for each respective dimension. Cluster centroids are formed from combinations of this value-set.

Additionally, a critical problem exists in using this approach with regards to removing division. This is demonstrated mathematically as follows:

$$P_d \leq \frac{\frac{S_i}{N_i} + \frac{S_j}{N_j}}{2} \tag{4}$$

where S_i, S_j represent two adjacent centroid accumulators in dimension d, and N_i, N_j represent their respective counters. To remove division, the expression would be written in this form:

$$2 * P_d * N_i * N_j \leq S_i * N_j + S_j * N_i \tag{5}$$

This new expression involves four more multiplication operations than the expression in Equation 3. This conflicts with one of the design decisions that we placed for the algorithm, mentioned earlier. Not only is the expression inefficient in terms of computations, but it also significantly inflates the bit-width of the comparators, especially for the left side of the expression.

When we discovered this we attempted a different method for building the tree, and it was to use the averages of two clusters for the internal node comparison values instead of the midpoints. In this method, internal node comparisons are done using this inequality:

$$P_d \leq \frac{S_i + S_j}{N_i + N_j} \tag{6}$$

When we remove division, this becomes the form of the inequality:

$$(N_i + N_j) * P_d \leq S_i + S_j \tag{7}$$

The advantage here is that this inequality has only one multiplication operation as well as only requiring two additional adders - this is not taxing on the hardware and is performance-friendly.

Once we had reached this stage in the method we implemented it and tested the quality of the clustering result. Unfortunately, it was not satisfactory and this can be seen in the histogram in Figure 14. The histogram shows the clustering output for the same image used to generate the histogram in Figure 12, also with 3 segments in each dimension. The most noticeable feature is that there is a large percentage of silhouette coefficient values that are negative, which indicates a high misclassification rate. The exact misclassification percentage is 14.6%, which is unacceptable based on our requirements. The average silhouette coefficient is 0.222, it is shifted heavily to the left due to its negative values.

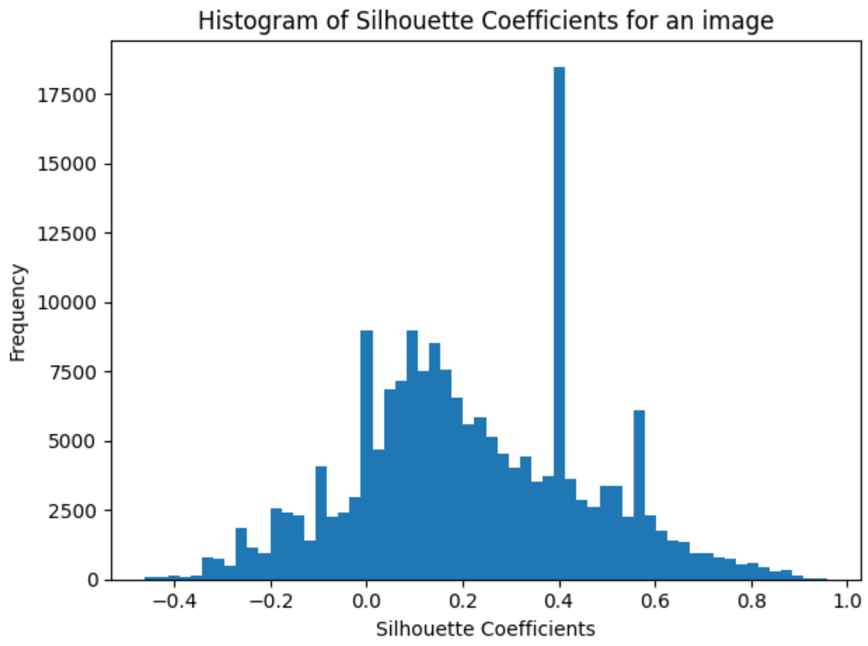


Figure 14: Silhouette coefficient histogram using the same image and same number of segments as in Figure 12, but with inequality 7 for internal tree comparisons. Many points are misclassified given the large quantity of negative values. The misclassification percentage is 14.6% and the average silhouette coefficient value is 0.222 .

The reason for these unsatisfactory results can be deduced after looking closer into the approach. It is based on doing three independent clustering runs on three different dimensions and then combining the results from each to form the final clusters. This method, for all intents and purposes, is an approximation so it is natural that we would not get the most optimal quality solution and our work was to determine whether it was a sufficient approximation. If we do not consider our division avoidance requirement it can be considered sufficient, but once we introduced another approximation through the utilization of inequality 7, the quality results deviated outside of our requirements. We had to abandon this approach and try something different.

5.6.2 KD-Tree Approach

The KD-tree approach was the final method that we attempted and was the one that worked out the best for us. Therefore we used it as the basis for our hardware algorithm. Many others have done similar work, with one of the earliest introductions of this done by Saegusa and Maruyama [11], with their hardware implementation of the filtering algorithm outlined by Kanungo et al. [7].

Winterstein et al. go further by improving upon the works of Saegusa and Maruyama by adding dynamic memory allocation as well as pipelining and parallelism [13]. Our work uses the basic idea behind Kanungo et al.'s filtering algorithm [7], but implements it in a slightly different way. We also go on to remove division from the kd-tree and introduce an inter-communication protocol for nodes within the tree.

The motivation behind using a kd-tree is primarily to reduce the required number of distance metric calculations and comparisons required between a point and the potential clusters. Abstractly, this is equivalent to filtering out clusters that are considered to be far away from the point so are not even considered among the potential nearest neighbour clusters. In practise, this is done using the kd-tree data structure which is shown in Figure 15. Each level corresponds to one of the possible data dimensions x, y, or z. A point enters the tree through the root and propagates downwards using a traversal technique that is based on depth-first search, but with a few differences. When a point arrives to a node, the distance metric (manhattan or euclidean) is calculated between the point and the node centroid. Additionally, the non-absolute axis component of the distance metric is recorded as a value d. The decision to first traverse left or right is then determined based on whether d is negative or positive respectively. Throughout traversal, the minimum distance value is kept track of as a variable *bestDist*. The cluster corresponding to the *bestDist* is also kept track of. When a point returns to the node from the first downwards traversal, the decision to make a second traversal is determined based on whether the value of d is less than the current value of *bestDist*. By the end of the traversal, the nearest cluster to the point will be identified and accumulated.

There is a significant performance advantage to using the kd-tree for clustering. The best case is when only one traversal is made for each visited node and this results in a complexity of $O(log(n))$. The worst case is when all nodes are visited, causing the method to devolve into a linear search and resulting in a complexity of $O(n)$.

This is very advantageous for data centers which might require require clustering for hundreds or possibly even thousands of clusters, so traversing all of these clusters will be too demanding on the computational resources. Additionally, in our method we introduce an experimental hyperparameter ε. ε is a user-defined constant that is added to d in order to increase the amount of cluster filtering. The greater the value of ε, the greater the number of filtered clusters, but also the larger the sacrifice in the search result accuracy.

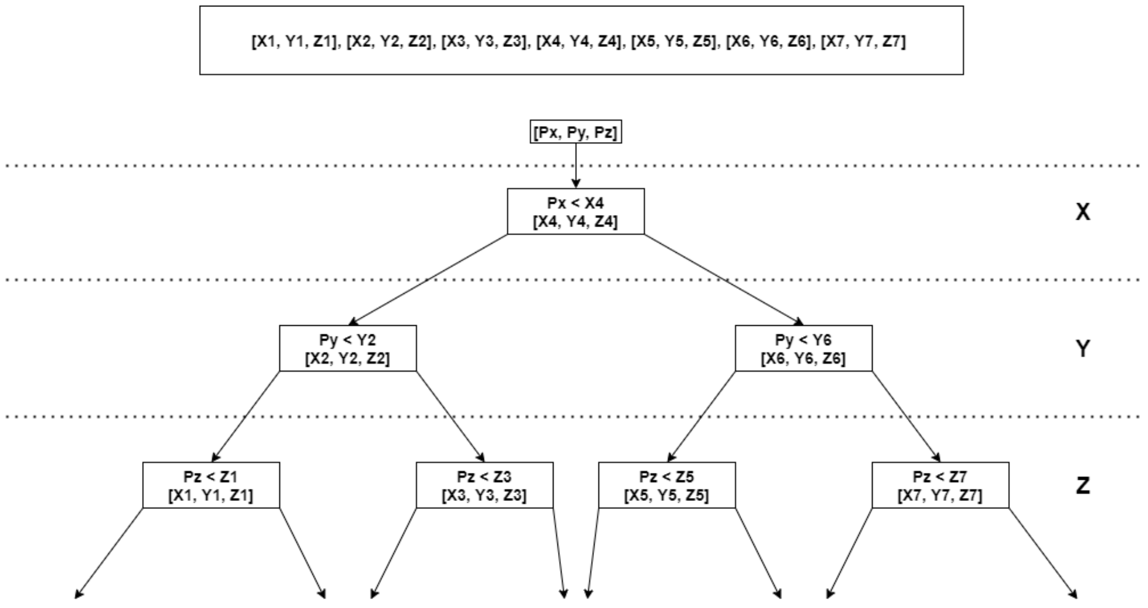


Figure 15: Kd-tree structure build from the sorted centroid array containing centroids 1 to 7. Each dimension down the tree corresponds to a different axis comparison between a centroid value in the current axis and the point value in the current axis. Points traverse the tree using depth-first search. The nearest cluster to the point is kept track of during traversal and is outputted once there are no more node traversals required.

Our hardware algorithm is split into three main stages: initialization, clustering, and binning. These are shown in algorithms 2, 3, 4 respectively. The main algorithmic stages use helper functions which are buildKdTree, traverse, clear, and updateStability. The helper functions are shown in algorithms 5, 6, 7, and 8 respectively.

The kd-tree method requires a distance metric and in our case we use the manhattan distance. To avoid division, since the manhattan distance is calculated from the centers we need an alternative equation that uses the accumulators and counters instead. This alternative equation is:

$$D_M = |P_x * N - S_x| + |P_y * N - S_y| + |P_z * N - S_z| \tag{8}$$

Where D_M is the manhattan distance, P_x, P_y, and P_z are the values of the point in each dimension, N is the cluster counter value, and S_x, S_y, and S_z are the respective accumulator values in each dimension. Internal node comparisons are performed using inequality 3.

Alternatively, the euclidean distance can be used and in this scenario the equation without division is as follows:

$$D_E^2 * N^2 = (P_x^2 * N^2 - 2 * N * P_x * S_x + S_x^2) + (P_y^2 * N^2 - 2 * N * P_y * S_y + S_y^2) + (P_z^2 * N^2 - 2 * N * P_z * S_z + S_z^2) \tag{9}$$

Where D_E is the euclidean distance. This is obtained by distributing this original equation which is as follows:

$$D_E^2 = (P_x - \frac{S_x}{N})^2 + (P_y - \frac{S_y}{N})^2 + (P_z - \frac{S_z}{N})^2 \tag{10}$$

Initialization The initialization technique that we used is the sub-box method outlined earlier in Figure 9. Algorithm 2 also shows the initialization of the global variables we will need throughout the algorithm. meansAccNew and pointCounterNew are used to compare with their respective old values to check for convergence. stabilityList is a list that checks for the stability of each cluster. Once all values of the stabilityList are 1, the algorithm ends as global convergence is reached. The threshold value is defined by the user (based on their required level of accuracy), and it is used for for convergence determination of the clusters. In the case of clustering with division, it is the minimum acceptable absolute difference between the current and previous iterations' centroid values. In the case of division removal, we use the threshold value to compare the current and previous iterations' accumulators and counters instead, since we do not compute the actual centroids. Lower threshold values result in more accurate globally converged cluster centroids, and typical values for it are around 2-4. We assume that the dataset is read prior to processing, and that it is stored in the data variable. This variable will be in memory in practise. Additionally, when we read the dataset we record the minimum and maximum values in each dimension (segMin and segMax), so that we may be able to compute the initial centroids. nSeg for each dimension is the user-defined number of segments in that dimension. The first step of the initialization is to find the cutpoints (value boundaries) in each dimension and to find the midpoints between them. These midpoints will form our initial segments. We get the initial centroids by forming the combinations of the initial segments. We also assign initial pointCounter values of 1 to all of the clusters to avoid division by 0.

Algorithm 2: Initialization

```
Global variables
meansAcc, meansAccNew, pointCounter, pointCounterNew,
stabilityList, kdTree
threshold = 2
data                                // represents the data points saved in memory

Initialization(x, y, z):
foreach $dim \in dimensions$ do
    cutpoints[nSeg + 1]
    initialValues[nSeg]                                          // declarations
    step = (segMax - segMin) / nSeg

    for $i = 0 \rightarrow nSeg + 1$ do
        cutpoints[$i$] = min + step*$i$
    end

    for $i = 0 \rightarrow nSeg$ do
        initialValues[$i$] = (cutpoints[$i$] + cutpoints[$i$+1]) / 2
    end

    foreach $i \in initialValuesX$ do
        pointCounter.append(1)
        pointCounterNew.append(1)
        foreach $j \in initialValuesY$ do
            foreach $k \in initialValuesZ$ do
                meansAcc.append($i$)
                meansAcc.append($j$)
                meansAcc.append($k$)
                meansAccNew.append($i$)
                meansAccNew.append($j$)
                meansAccNew.append($k$)
            end
        end
    end
end
```

Clustering Clustering is the processing stage of the algorithm that generates the clustered model. Clustering occurs until global convergence is reached, which is kept track of using the stable variable. Clustering happens in iterations and the goal is to reduce the number of iterations as much as possible. This allows for a faster global convergence and reduces the required processing. At the beginning of each iteration the kd-tree is built using *buildKdTree* and is stored in the kdTree variable. kdTree is accessed through its root. Next, the meansAccNew and pointCounterNew variables are cleared using *clear*, to begin an accumulation stage. Next, the points in the dataset are fed into the tree for traversal using *traverse*, to find the nearest cluster to each of them so that it may be accumulated. Once all of the points are accumulated, local convergence is checked for each cluster using *updateStability*. The stable variable is then computed by ANDing the values of the stability list. Clustering ends once stable = 1.

Algorithm 3: Clustering

```
Clustering():
stable = 0
while not stable do
    kdTree = buildKdTree(meansAcc, pointCounter, 0, indices(pointCounter)
    for $i = 0 \rightarrow length(pointCounterNew)$ do
        if $pointCounterNew[i] \neq 0$ then
            clear(i)
        end
    end

    foreach $point \in data$ do
        clusterIndex = traverse(kdTree, point, 0, -1, 0)
        meansAccNew[clusterIndex] += point                 // 3 accumulations
        pointCounterNew[clusterIndex] += 1
    end

    for $i = 0 \rightarrow length(pointCounterNew)$ do
        updateStability(i)
    end

    stable = &[stabilityList]                              // ANDing all of the list
end
```

Binning Once global convergence is reached, binning can occur. Binning can also occur separately based on our second use case where the user inputs a point based on a pre-trained clustered model. Binning occurs exclusively using the *traverse* method. The cluster index is then given as an output.

Algorithm 4: Binning

```
Binning():
foreach $point \in data$ do
    clusterIndex = traverse(kdTree, point, 0, -1, 0)
    output(clusterIndex)
end
```

Building the Kd Tree This method builds the kd-tree at the beginning of each iteration to prepare it for point traversal. Since we avoid calculating centroid values (due to avoiding division), we use the cluster accumulators and counters instead as the values of the tree nodes. Indices are also stored in within the node so that they can be returned so that their corresponding clusters can be accumulated. The axis value is propagated down the tree so that the correct computation of the single-dimension distance value d can be performed. Therefore, axis can only be 0, 1, or 2 corresponding to the x, y, and z dimensions respectively. The first step is to sort the cluster accumulators based on their values in the current axis. The axis is updated for the next dimension and is propagated down the tree. The central cluster index is then determined. The left half the sorted clusters up to the central cluster is then assigned the the current node's left connection, and the right half after the central cluster is assigned to the right connection. The current node's internal value D is assigned the central cluster. D will contain three values which are the accumulators, the counter, and the index. Tree building is done recursively all the way down till only one cluster remains as the central cluster, and this central cluster will represent a leaf node. Leaf nodes have NULL left and right connections. In hardware, instead of recursive tree-building a user-defined maximum depth tree will be synthesized and its internal node values will be updated at the beginning of each iteration.

Algorithm 5: Building the Kd Tree

```
buildKdTree(accumulators, counters, axis, indices):
if length(counters) > 1 then
    sort(axis)         // sort accumulators, counters, and indices based on axis
    axis = (axis + 1) % 3
    half = length(counters) / 2
    x = node()      // nodes have left and right connections, as well as center
     values that represent their cluster
    x.left = buildKdTree(accumulators[:half*3], counters[:half], axis, indices[:half])
    x.right = buildKdTree(accumulators[half*3:], counters[half:], axis, indices[half:])
    x.D = [accumulators[half*3], counters[half], indices[half]]
    return x
end
else if length(counters) == 1 then
    x.left = NULL
    x.right = NULL
    x.D = [accumulators[0:3], counters[0], indices[0]]
    return x                                                        // leaf nodes
end
```

Kd Tree Traversal Once a tree is built, points must traverse through it so that they can be assigned to their correct nearest clusters, and this is done with *traverse*. One efficiency improvement that we make in our implementation is the introduction of the user-defined hyperparameter ε (epsilon). ε is an additive margin for the single-dimensional distance that reduces the number of tree traversals for points. The tradeoff is that the misclassification percentage is increased by a small margin. The nodes that the points traverse through have left and right connections, as well as a value which is represented with D. D is a list of size 3 and stores the xyz accumulators at index 0, the counter at index 1, and the cluster index at index 2. The first step during traversal is to calculate the distance metric, which in our case is the manhattan distance. This is calculated using Equation 8. Next, the single-dimensional distance value d is calculated based on the current axis value. If no prior value for bestDist was assigned, it is assigned the value of the current node distance (root node). If the current dist value is less then the value of bestDist, bestDist is updated with the dist value and bestIndex is updated to indicate that the current cluster is closer to the point. Axis is then updated. Next, a binary value otherBranch is determined. otherBranch is an indicator of whether the other branch should also be traversed to, after visiting and returning from the first branch. This is based on a depth-first search approach. The first branch that should be traversed to is determined based on the sign of d. If d is negative then the point traverses left, and if it is positive the point traverses right. The binary value firstDirection is determined based on the direction of the first traversal. A left-first traversal corresponds to a firstDirection value of 0, and a right-first traversal corresponds to a firstDirection value of 1. If otherBranch is 1, the second direction is also traversed to based on the prior determination of the value of firstDirection. The algorithm ends once traversal is complete and the bestIndex is returned.

Algorithm 6: Kd Tree traversal

```
epsilon = 50    // epsilon is a user-defined hyper-parameter that specifies an
 additive margin for traversing one branch of the tree
traverse(node, point, axis, bestDist, bestIndex):
  // node.D[0] => xyz accumulators, node.D[1] => counter, node.D[2] => index
if $node \neq NULL$ then
    dist = $|node.D[1] * point[0] - node.D[0][0]| + |node.D[1] * point[1] - node.D[0][1]| +$
     $|node.D[1] * point[2] - node.D[0][2]|$    // Manhattan distance without division
    d = node.D[0][axis] - point[axis]*node.D[1]    // Distance in the axis dimension
     without division
    if $bestDist == -1$ or $dist < bestDist$ then
        bestDist = dist
        bestIndex = node.D[2]
    end
    axis = (axis + 1) % 3
    otherBranch = $|d| + epsilon < bestDist$
    if $d < 0$ then
        traverse(node.left, point, axis, bestDist, bestIndex)
        // We traverse either the left or right branch depending on whether d
           is positive or negative
        firstDirection = 0
    end
    else
        traverse(node.right, point, axis, bestDist, bestIndex)
        firstDirection = 1
    end
    if otherBranch then
        if not firstDirection then
            traverse(node.right, point, axis, bestDist, bestIndex)
        end
        else
            traverse(node.left, point, axis, bestDist, bestIndex)
        end
    end
    return bestIndex
end
```

Clear At the start of a new iteration, the meansAccNew and pointCounterNew variables for all of the clusters need to be cleared and this is done with *clear*. First however, meansAccNew is assigned to meansAcc and pointCounterNew is assigned to pointCounter. This is because they are needed for tree building and traversal, as well as to check the local convergence condition for each cluster. After saving them, they are cleared so that they can be used for the upcoming iteration.

Algorithm 7: Clearing cluster accumulators and counters

```
***clear(index):***
meansAcc[index*3:index*3 + 3] = meansAccNew[index*3:index*3 + 3]
pointCounter[index] = pointCounterNew[index]
    // saving the accumulator and pointer values from the previous iteration
meansAccNew[index*3:index*3 + 3] = [0, 0, 0]
pointCounterNew[index] = 0
```

Update Stability *updateStability* is used to check local convergence, or stability, for each cluster. This is checked through the accumulator values in each dimension. Alternatively, the pointCounter value can also be checked but this might be unnecessary. If the absolute difference between meansAccNew and meansAcc is below the user-defined threshold, then the cluster's stability value within the stabilityList is set to 1. Clustering ends once all of the values within stabilityList are 1.

Algorithm 8: Updating the stability list for each cluster

```
***updateStability(index):***
if $|meansAccNew[index * 3] - meansAcc[index * 3]| < threshold$
***and***
$|meansAccNew[index * 3 + 1] - meansAcc[index * 3 + 1]| < threshold$
***and***
$|meansAccNew[index * 3 + 2] - meansAcc[index * 3 + 2]| < threshold$ then
    stabilityList[index] = 1
end
```

Software results The verification results from the kd-tree method implemented within our golden model were highly satisfactory. This can be seen visually in Figure 16 below. By comparing this output with the one obtained in Figure 13, we can see that this method is much better in terms of visual clustering quality.

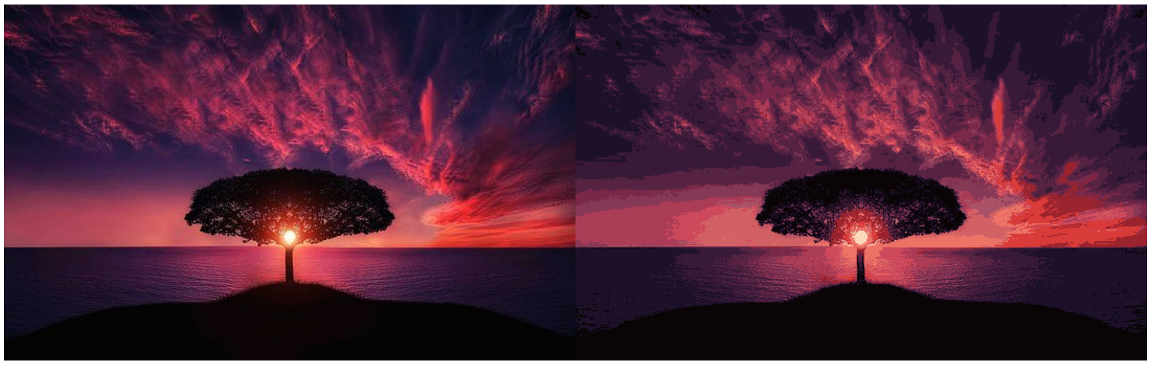


Figure 16: Side by side comparison of the input image on the right and the segmented output using kd-tree clustering. It can clearly be observed that the clustering quality from the segmented image obtained using this method is much better than the clustering quality of the segmented image obtained using the dimension separation method, which was shown in Figure 13.

Figure 17 shows the silhouette coefficient histogram for the output image in Figure 16. The average silhouette coefficient is 0.536 and the misclassification percentage is 4.2%. This is for an initialization using three initial segments in each dimension. This further verifies that the kd-tree method satisfies our quality requirements as the silhouette coefficient values are mostly very high, and the misclassification percentage is very low (below 5%). The reason for the misclassification is due to some of the correct clusters being filtered out during point traversal, but this happens very rarely and is the trade-off for obtaining high efficiency. Misclassification is mostly apparent with low data ranges and this is the case with images. Although it should also be noted that the negative silhouette coefficient values are very low with most of them below 0.2 . This indicates that the misclassified cases happened between clusters that were very close to each other which is often an acceptable case. the misclassification tends to be between 0 and 1 for data ranges of 1000 or more.

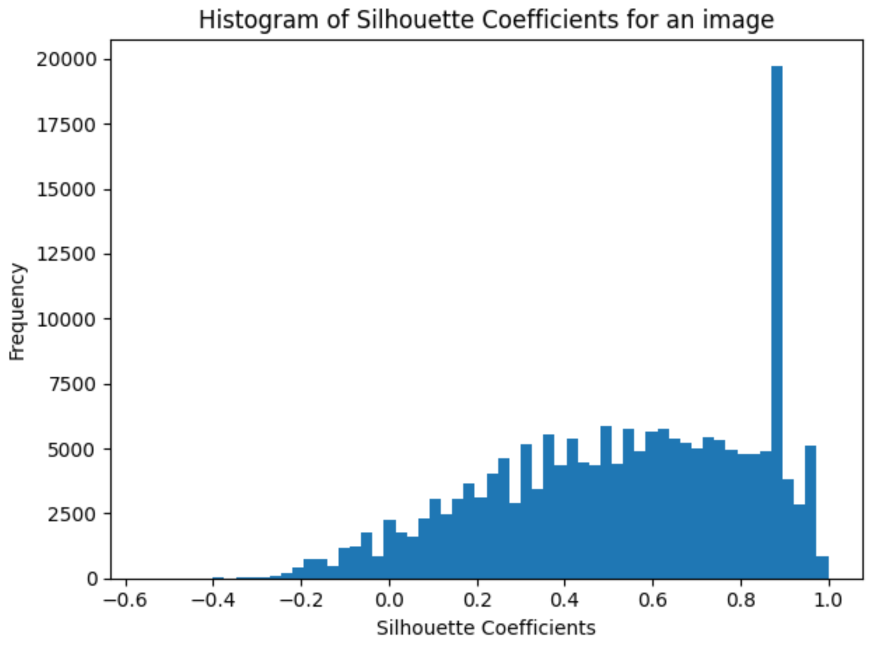


Figure 17: Silhouette coefficient histogram for the kd tree method corresponding to the output image in Figure 16. The initial number of segments is 3 in each dimension. The average silhouette coefficient is 0.536 and the point misclassification percentage is 4.2%.

Additionally we tested the algorithm with 1000 clusters for randomly generated data, the average number of node visits was 60.935, which for n = 1000 is equal to $6 * log_2(n)$.

Once we recognized the efficiency and quality of the kd-tree method we went on to implement it in hardware.

6 Component's Design and Micro Architecture

Coming into the hardware circuit design we had some guidelines that we needed to follow. The first of these is that we did not want to use any global variables. This is because global variables will have large fan-outs which is wasteful of resources and not a good design practise. Second, the number of nodes within the kd-tree needed to be configurable at runtime. This is to allow the user to edit their own parameters to get a tailored clustering result. Finally, we wanted to allow more than one tree to be generated if this is allowable to provide parallel processing capabilities. With that being said, we came up with a design that uses four modules in a hierarchical fashion. These are the node, the processing element, the comparison element module, and the distance module. In practise, the processing element is part of the node but here we explain its functionality as an abstraction and consider that it is its own independent module. The KD tree is made up of nodes, and each node contains a processing element and a comparison element. Comparison elements contain a distance module. We provide the option to use either the manhattan distance or the euclidean distance for the distance module. Note that for simplicity, the design is explained assuming that we use division because this does not alter the circuit's functionality. In practise, the division avoidance is implemented and we perform this using inequality 3, and equations 8, or 9. All of the verilog circuits were written, synthesized, and tested in Intel Quartus Prime. The Cyclone V board was used for synthesis.

6.1 KD Tree Structure in Hardware

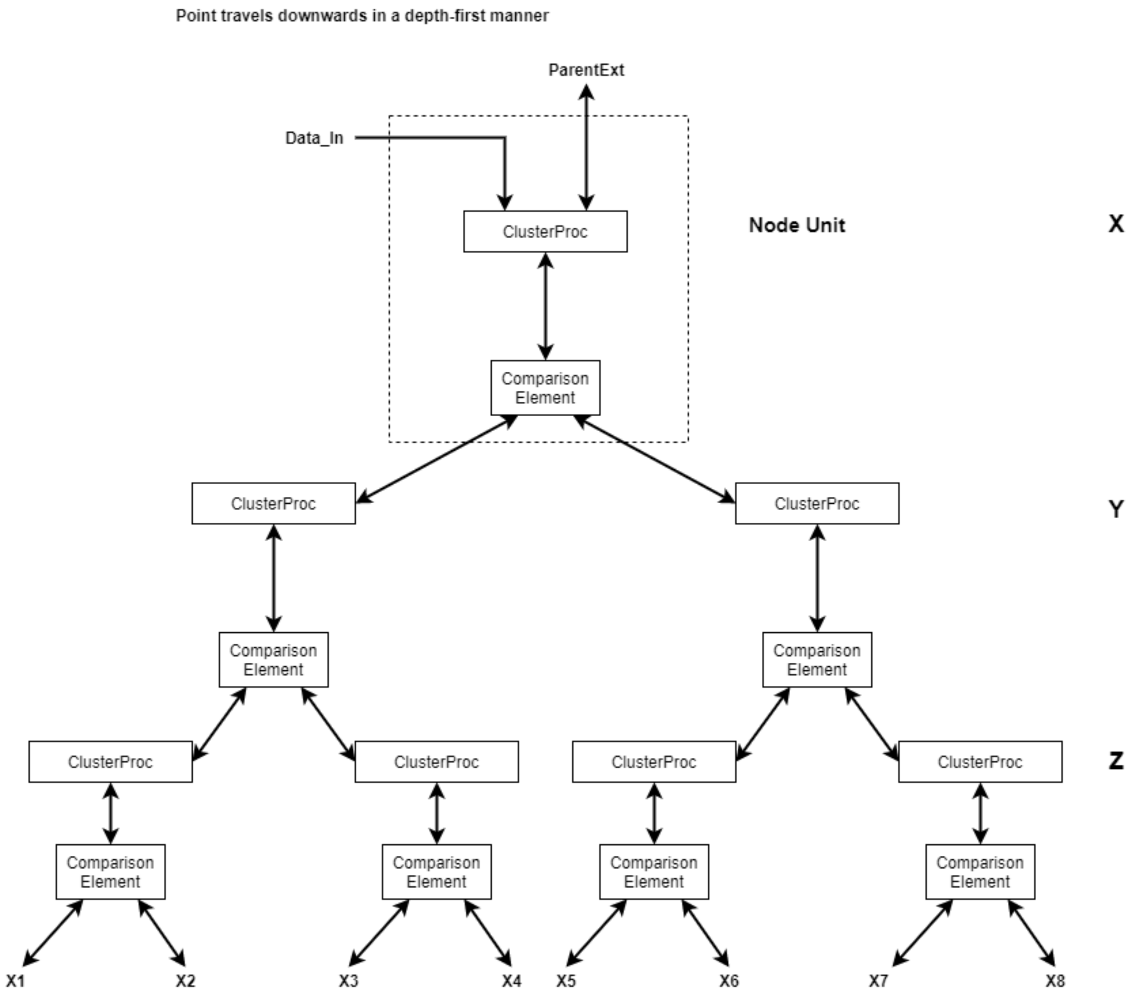


Figure 18: Hardware structure of the kd-tree. The kd-tree is built from nodes and each node contains a cluster processor element and a cluster comparison element.

The kd-tree hardware architecture, which is shown in Figure 18 is built so that is made up of individual nodes where each node represents a cluster. The ParentExt and X1 to X8 connections represent potential node connections to an external parent, and external children nodes. In practise however, the tree will have a fixed maximum size (depending on the available hardware resources), and the top-level node is the root and it is the tree's access point. Data moving through the tree can travel in a bidirectional fashion depending on a node's current stage. The main algorithm stages are initialization, sorting, and point propagation. The algorithm stages are outlined in greater detail in Section 6.2. The processing and comparison elements will have different functions depending on the current stage, as well as the commands being received to them from adjacent nodes. Transitioning from one stage to another happens when a stage's end signal arrives to the root. This end signal is detected, triggering the next stage's signal that is given as a command back to the root. The control circuit that triggers stage signals into the tree, and receives acknowledgements back from the tree is referred to as the controlling circuit. The controlling circuit communicates directly to the tree's root.

To start the clustering process and begin the initialization stage, the center_fill command is given to the root and the initial clusters are entered through data_in. Once all of the nodes have centers in them, a center_fill_done command arrives back to the root triggering the sort signal. This causes a transition to the sorting stage.

Sorting happens in multiple steps to ensure that each node is sorted within its own respective axis. In other words, the first step is to fix the root by making sure it is the central value along the x-axis. Then the next step is to fix the root's left and right children, by making sure they are the central values along the y-axis and between all nodes along the left and right side of the root respectively. Bear in mind nodes to the left and right of the root are sorted independently from one another. This sort propagation occurs all the way to the tree's leaves and when the whole tree is sorted, it transitions to the point propagation stage.

An input point arrives through data_in during point propagation and is sent to the comparison element. The comparison element decides whether to send the point to the left or to the right depending on the value of single-dimension distance value d. The point will then traverse downwards while the bestDist value and the cluster corresponding to it are kept track of. The point will eventually return back to the original node that sent it in a specific direction. The node's comparison element will then determine whether the point needs to be sent to the other direction by comparing the values of d and bestDist. Eventually, the point will return to the root with the nearest cluster to it. The centroid value of this cluster will then propagate downwards in a manner similar to broadcasting, and the cluster with a centroid equal to it will be accumulated. Accumulation is done within the cluster processing element. Once all points have been accumulated, the cluster centroids will be updated accordingly and a convergence check will be performed. If at least one cluster has not converged, the signal to begin the next iteration is triggered. Note that sorting needs to occur before each iteration because the updated clusters might have shifted and need to be rearranged correctly. Otherwise if all of the clusters have converged, clustering ends and the final converged clusters can be found within their respective nodes.

6.2 Algorithm Stages in Hardware

6.2.1 Initialization

In this stage, the node centers are initialized by being propagated from the root to the all of the inner nodes until the leaves. The initial centers arrive to the root in a serial stream. Once all of the leaves are filled and have centers within them, they send out center_fill_done signals to their parents. These signals propagate all the way to the root, which will in turn send it to the controlling circuit. The controlling circuit will then send out a start_sorting_as_root signal to move to the sorting stage.

6.2.2 Sorting

Sorting is an essential stage that prepares the tree and fixes its centers in preparation for the point propagation stage. In our design we introduce a novel technique for sorting down the tree.

The advantage to this technique is that non-adjacent node triplets can sort in parallel. This happens through node comparisons with their neighbours until all nodes have stabilized. Sorting stability between three nodes is achieved if the inequality Left < Parent < Right is satisfied. Additionally, each inner node within the tree must ensure that it is greater than the rightmost leaf node below it, and less than the leftmost leaf node below it. Once these two inequalities are satisfied for the whole tree, the sorting stage can end. Sorting happens in multiple sub-stages and with varying axes down the tree. The number of sub-stages is equal to the number of levels within the tree. The first level is sorted and the root node is centered with respect to the X axis component, then the second level is sorted and the root's children are centered with respect to their Y axis components, then the third level is sorted and the next children are centered with respect to their Z axis components, all the way down until the leaf nodes are reached. At the beginning of each sort sub-stage, the sorting axis is propagated downwards and this can be seen in Figure 19. Figure 20 shows how the sorting axes vary with each sub-stage that alternate with each descent down the tree. Figures 21, 22, 23, and 24 show the detailed steps that occur in a single sorting sub-stage along the X axis. The red boxes indicate the sorting operations that occur at the same time as each other, and the arrows indicate the signals that coordinate comparison interactions between nodes. Note that not all signals are shown in these diagrams for the sake of clarity.

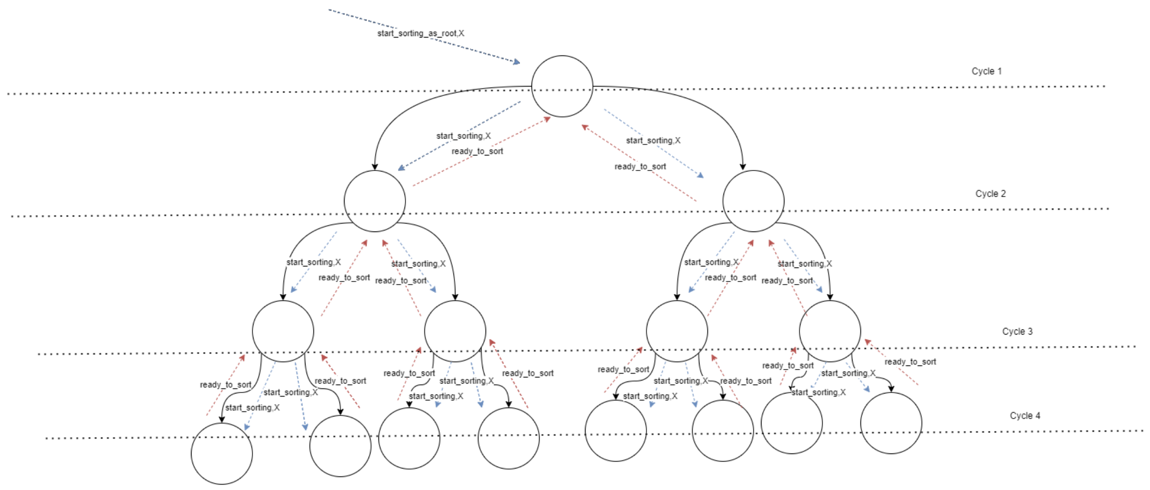


Figure 19: Initializing the sorting axis down the tree.

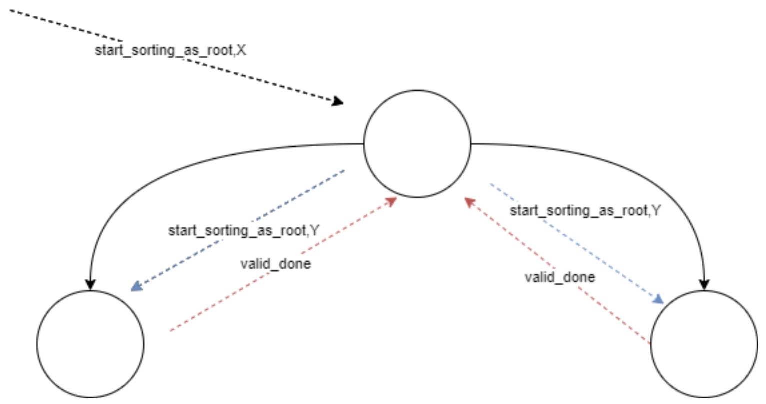


Figure 20: After sorting in one axis is complete for a node and its center is fixed for that axis, it makes its direct children virtual roots for sorting so that they may also be fixed for the next axis's sorting stage.

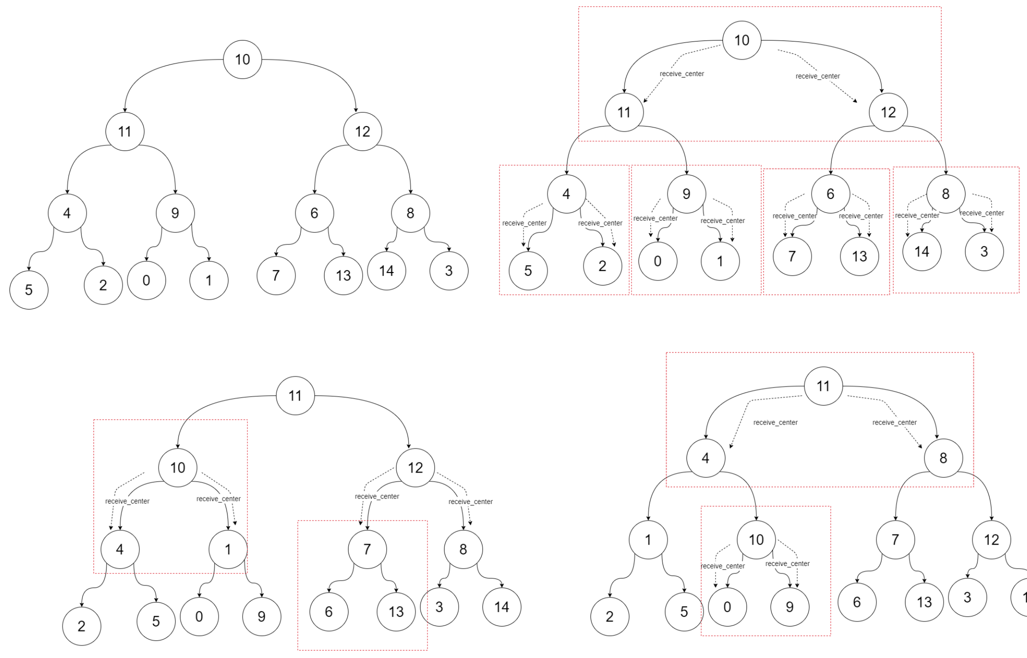


Figure 21: Sorting stage example, steps 1-4.

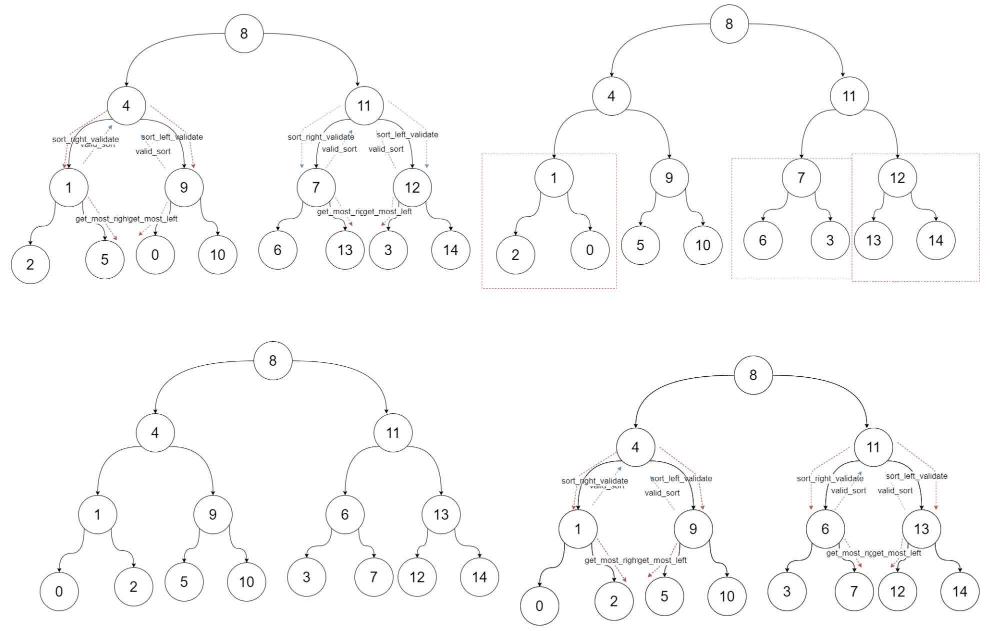


Figure 22: Sorting stage example, steps 5-8.

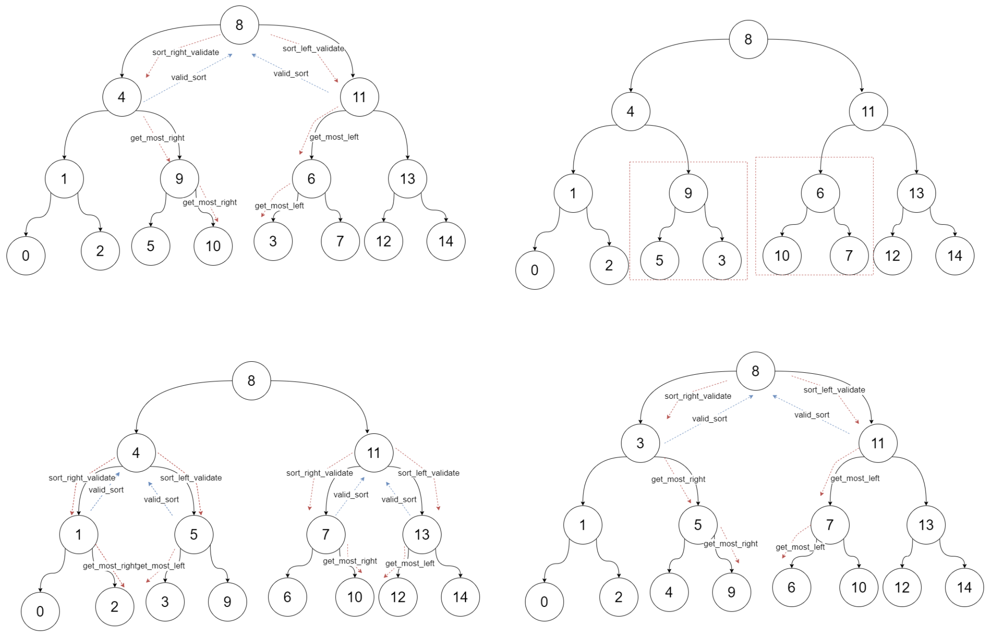


Figure 23: Sorting stage example, steps 9-12.

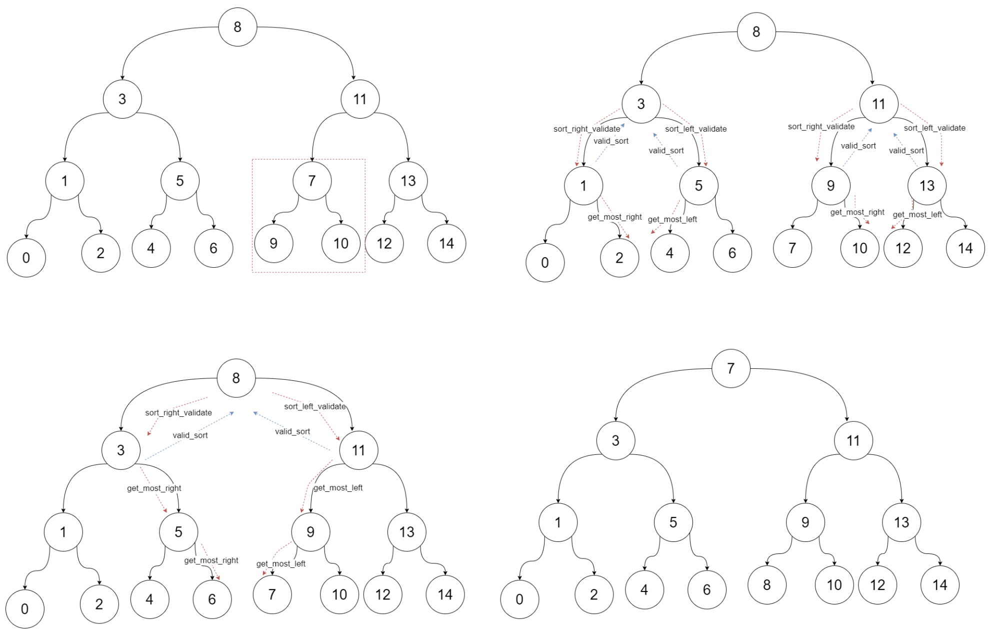


Figure 24: Sorting stage example, steps 13-16.

6.2.3 Point Propagation

Point propagation is the main stage that occurs during clustering, and the purpose of it is to find the nearest cluster to a point based on the distance metric, so that this resultant cluster may be accumulated with the point. In our design, this happens in a depth-first based traversal. While the point is propagating downwards, the best cluster value is kept track of. At any given node, if the current best cluster distance is smaller than the distance component in a given axis of the node, then the other branch of the node is not visited. This is the basis of the filtering technique and it is highly effective in reducing the search space. Figure 25 shows an illustration of how this occurs in practise.

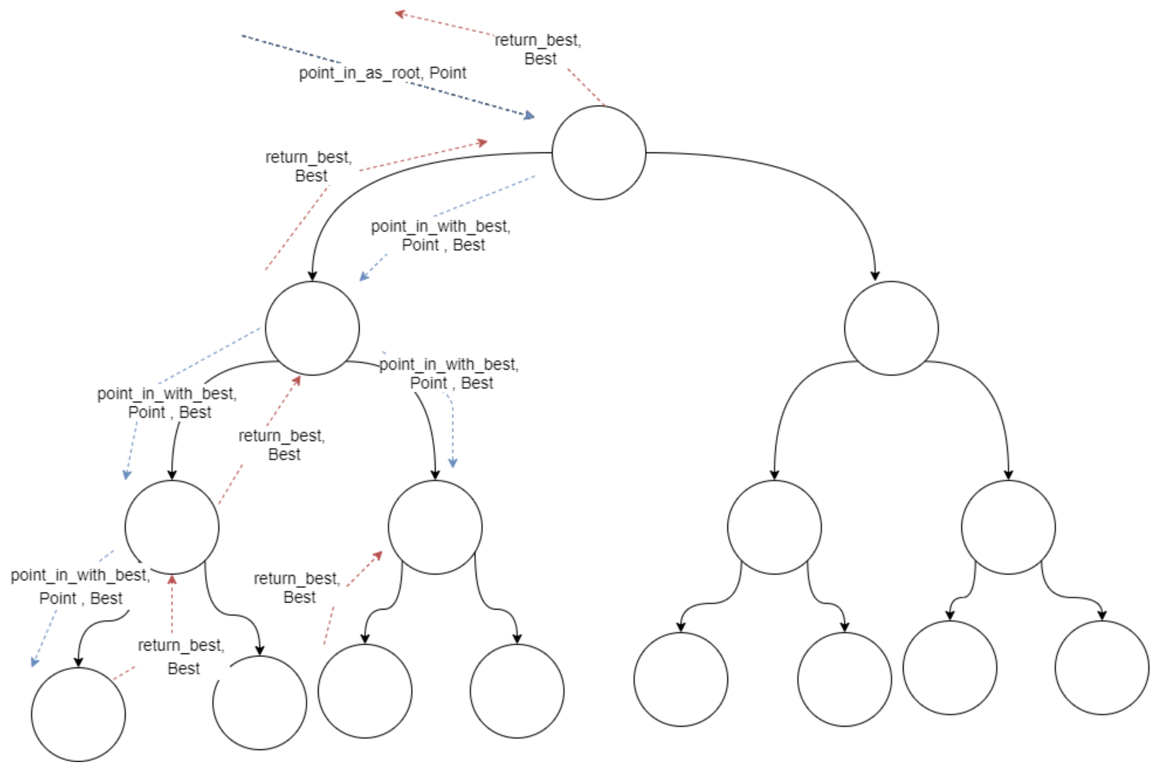

return_best, Best
point_in_as_root, Point
return_best, Best
point_in_with_best, Point , Best
point_in_with_best, Point , Best
point_in_with_best, Point , Best
return_best, Best
point_in_with_best, Point , Best
return_best, Best
return_best, Best

Figure 25: An example of the traversal sequence that occurs during point propagation. Note that filtering is also shown here as the entire right side of the tree was not visited.

6.3 Node - Command Handler

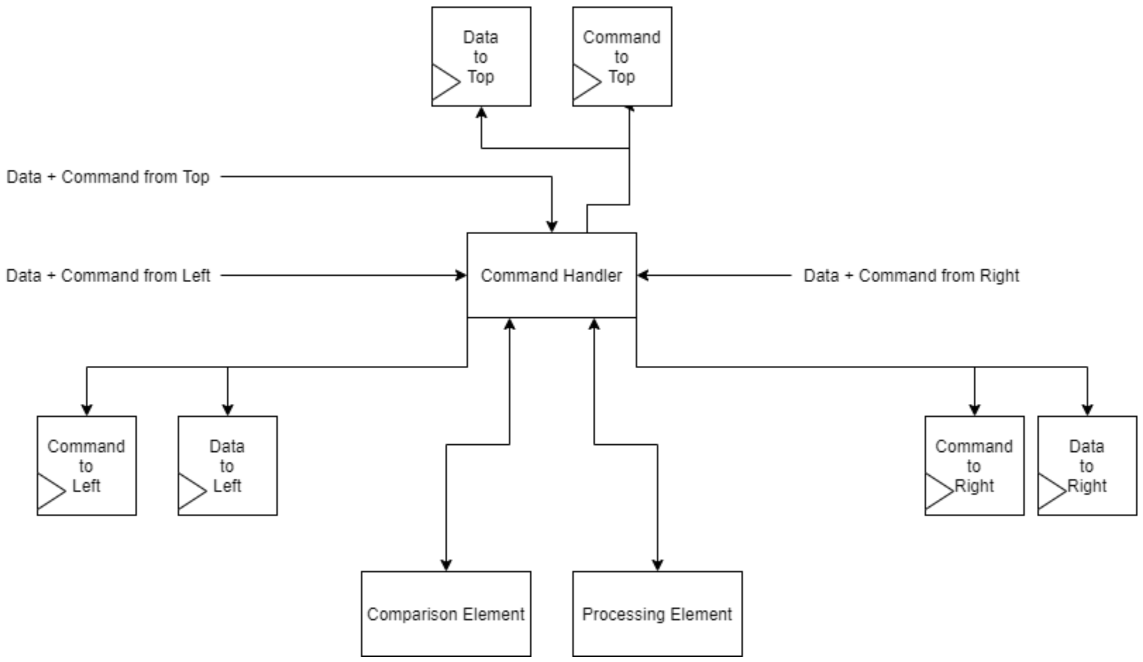


Figure 26: Cluster node architecture. The cluster node contains the command handler, the processing element, and the comparison element. It also has register connections to its left and right children as well as its parent. For each of these connections, it contains a data and command register.

6.3.1 Design

The node module, which is shown in Figure 26, is the building block of the kd-tree and functions based on the commands it receives which are dealt with in the command handler. Therefore, the command handler is acts as the control unit for the node. Additionally the command handler can send outgoing commands to adjacent nodes. The command handler is not a separate module but instead is the central intrinsic control element within the node, and can also be considered to be the node itself.

Commands and data can come from and go to four different places. These are the parent node, the left-connected node, the right-connected node, and the node itself. To avoid control hazards, we enforce a priority scheme for functional command flow. This scheme defines that the order of descending priority of arriving commands to be dealt with is as follows: the node's parent, the node itself, the left-connected child, and the right connected child. This means that if two or more conflicting commands arrive to the node, the only command that will be executed is the one with the highest priority and all other commands will be ignored. A code snippet of how this is implemented in hardware is shown below:

```
always @(posedge clk) begin
    if(command_from_top != nop)
        case(command_from_top)
            command 1:
            ....
            command n:
        endcase
    else if (self_command != nop)
        case(self_command)
            command 1:
            ....
            command n:
        endcase
    else if (command_from_left != nop)
        case(command_from_left)
            command 1:
            ....
            command n:
        endcase
    else if (command_from_right != nop)
        case(command_from_right)
            command 1:
            ....
            command n:
        endcase
end
```

We define a set of 32 possible commands that the command handler accepts and sends, that form the complete basis of the node's functionality. Each of these commands has a unique opcode corresponding to it. Table 3 below shows these commands with their functionality. The table also includes columns indicating propagation (P), acknowledgement (A), and whether the signal is internal or external (I/E). Propagation indicates that the signal propagates downwards within the tree. Acknowledgement indicates that the signal has a corresponding acknowledgement signal that must return to it from below. Internal signals are signals that are generated from within the tree as a result of node interactions, and only stay within the tree. External signals are ones that are issued from the controlling circuit.

Table 3: All possible node commands and their functionality. The propagation (P) column indicates whether the signal propagates downwards. The acknowledgement (A) column indicates whether the signal requires an acknowledgement. The I/E column indicates whether the signal is internal to the tree, or externally arriving from the controlling circuit.

Command	**P**	**A**	**I/E**	**Description**
rst	Y	Y	E	Triggers the reset mechanism in the node. Receiving node will propogate this command to its children. It is sent by the controlling circuit.
rst_done	N	N	E	Triggered when all lower nodes have been reset. It is the acknowledgement signal corresponding to rst and is received by the controlling circuit.
center_fill	Y	Y	E	Initializes the node centroid. Node centroid initialization down the tree is done serially. Serial input centroids are sent by the controlling circuit.
center_fill_done	N	N	E	Triggered when all lower nodes have been initialized. It is the acknowledgement signal corresponding to center_fill and is received by the controlling circuit.
switch	N	N	I	Internal command used in the sorting stage and is triggered by a parent node within itself when it detects that the condition left < parent < right has been violated.
receive_center	N	N	I	Internal signal between nodes in the sorting stage used to exchange centers.
busy	N	N	I	Internal signal used in the sorting phase while exchanging data. Prevents the interruption of the data exchange.
dne	N	N	I	Used to indicate the absence of a child connection (either a left or a right child).
start_sorting	Y	Y	E	Sent by the controlling circuit to signal the beginning of the sorting stage.
sort_done	N	N	E	Triggered when all nodes under the root have been sorted. It is the acknowledgement signal corresponding to start_sorting.
ready_to_sort	N	N	I	Internal command used in the sorting stage to signal that the sending node is ready to switch centers with its neighbours.
sort_left_validate	N	Y	I	Internal command used in the sorting stage by the parent of all nodes below it to make sure that its value is greater than its left-most child.
sort_right_validate	N	Y	I	Internal command used in the sorting stage by the parent of all nodes below it to make sure that its value is less than the right-most child.
valid_sort	N	N	I	Internal command used to signal that all lower nodes are sorted.

valid_done	N	N	I	Internal command sent by a child to its parent as a result of it having received sort_left_validate and sort_right_validate from its own children.
next_sort_level	N	N	I	Internal command used after the completion of sorting in one level to prepare the tree for sorting in the next level.
start_sorting_as_root	N	N	E	Internal command used to set a node as the root of a sorting stage. The number of sorting roots is doubled with each step down tree.
point_in_as_root	N	Y	E	External command sent to the root node by the controlling circuit to cluster a point.
point_in_with_best	Y	Y	I	Internal command used while propagating the point and the current closest cluster in the tree.
return_best	N	N	E	Acknowledgement signal for point_in_as_root and point_in_with_best. The return value of this command is the current closest cluster to the point.
hold	N	N	I	Internal command used in the sorting stage by a node to freeze the centers of its children, when it needs to switch centers with its parent. This is to prevent a switching conflict between a parent-self switch and a self-child switch.
get_most_left	Y	Y	I	Internal command used in the sorting stage to return the left-most child's center to the sender node.
get_most_right	Y	Y	I	Internal command used in the sorting stage to return the right-most child's center to the sender node.
set_most_left	Y	N	I	Internal command used in the sorting stage to set the left-most child's center to a specific value.
set_most_right	Y	N	I	Internal command used in the sorting stage to set the right-most child's center to a specific value.
nopme	Y	N	E	Used to force all receiving nodes to set their command_to_top to nop. This command must be used by the external circuit after the clustering of a point.
axis_set_inc	Y	N	E	Sent by the external circuit to set the root's axis. It is used if the sorting stage is skipped and the tree is already sorted correctly. This is useful when only binning is required.
accumulate	Y	N	E	Internal command sent by the root to all nodes. Only the node with the same center as the arriving data will be accumulated.
divide	Y	Y	E	Command sent by the controlling circuit to divide the accumulators and counters of each node. The division result is stored in the new_center register.
new_iteration	Y	N	E	Command issued by the controlling circuit to begin a new iteration. This command will cause all nodes to replace their center values with new_center and then clear the new_center value.

stable	N	N	E	Command received by the controlling circuit to indicate that the kd-tree is stable - global convergence.
unstable	N	N	E	Command received by the controlling circuit to indicate that the kd-tree is unstable - has not reached global convergence.

6.3.2 Verification

Verification of the node module was done by verifying the kd-tree as a whole. This is because the functionality of the node module is dependant on its relationship to its adjacent nodes as well as the current stage in the algorithm. Therefore, it has been verified in the verification of the kd-tree structure in the kd_tree.v testbench module. To generate the instantiation text for the testbench a python program called 'tree_generator.py' was developed. This program takes the input number of nodes and returns the instantiation text with all of the necessary connections between these nodes, so that it can be entered into the testbench accordingly. Debug messages were also inserted at different stages within the actual node module so that analysing its operation can be more convenient. The testbench was run on ModelSim and the waveforms were observed. The input used was a standard image in hex format. The clustering output was then written to a text file and was compared against the same output produced using the golden model. Once the two outputs were seen to be the same, the circuit's operation was deemed to be verified and correct.

6.3.3 Specifications

- 340 ALUTs.
- 139 registers.
- 297 LUTs.
- 32 possible commands.
- 5 bit command opcode.

6.4 Processing Element

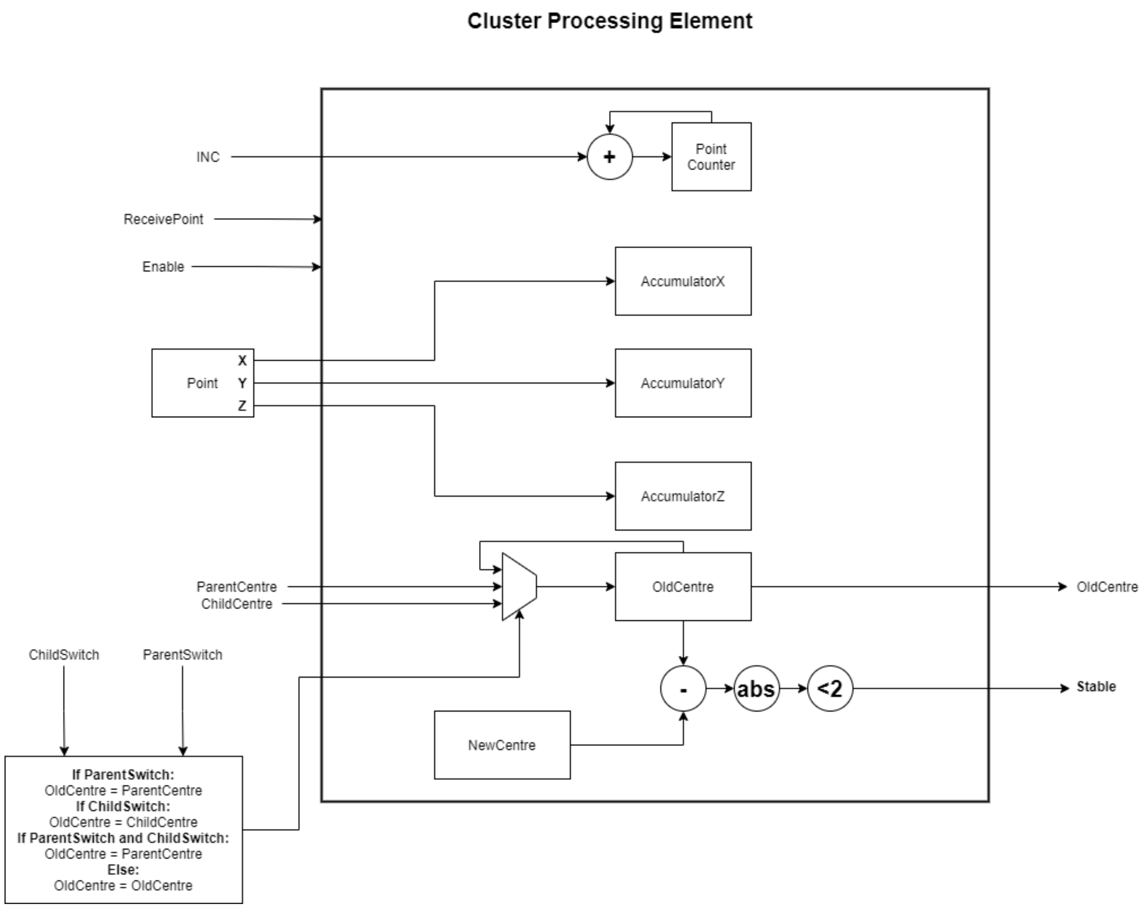


Figure 27: Architecture of the cluster processor. The cluster processor contains the old and new center values, the xyz accumulators, and the point counter. Local convergence, or stability, is checked from within the cluster processor.

6.4.1 Design

The cluster processing element, which is shown in Figure 27, is responsible for maintaining the registers belonging to cluster within the node. These variables are the xyz accumulators, the point counter, and the old and new centers. The cluster processor also receives points so that is may accumulate them. Moreover, it receives childSwitch and parentSwitch commands for switching with parent or child during the sorting stage. The cluster processor shown here is an abstraction for simplicity. In practise, everything contained within this module can be accessed directly from within the node.

6.4.2 Verification

The cluster processor was verified using a individual testbench, which was cluster_pe_tb.v . The reason for doing this was that the cluster processor can work independently if need be, so its functionality can be tested independently as well. Since this functionality is also rather primitive, comparing it with a separate golden model was not required. Verifying this module was a matter of ensuring that it performs as required when it receives certain input signals. Operation was eventually seen to be correct as is specified by the module's required functionality in certain stages.

6.4.3 Specifications

- 72 registers.
- 4 LUTs.

6.5 Comparison Element

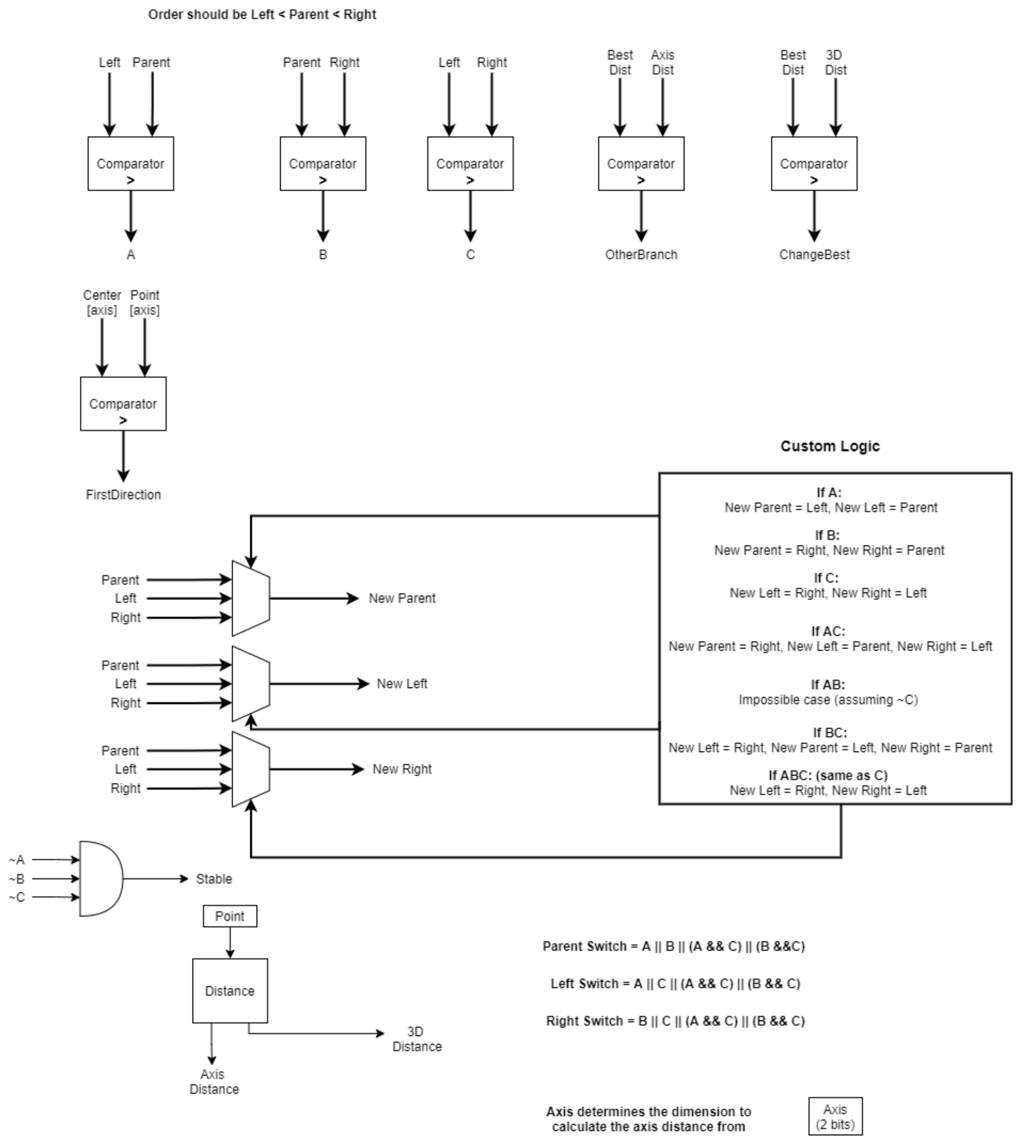

Order should be Left < Parent < Right
Left
Parent
Comparator >
A
Parent
Right
Comparator >
B
Left
Right
Comparator >
C
Best Dist
Axis Dist
Comparator >
OtherBranch
Best Dist
3D Dist
Comparator >
ChangeBest
Center [axis]
Point [axis]
Comparator >
FirstDirection
Custom Logic
If A:
New Parent = Left, New Left = Parent
If B:
New Parent = Right, New Right = Parent
If C:
New Left = Right, New Right = Left
If AC:
New Parent = Right, New Left = Parent, New Right = Left
If AB:
Impossible case (assuming ~C)
If BC:
New Left = Right, New Parent = Left, New Right = Parent
If ABC: (same as C)
New Left = Right, New Right = Left
Parent
Left
Right
New Parent
Parent
Left
Right
New Left
Parent
Left
Right
New Right
~A
~B
~C
Stable
Point
Distance
3D Distance
Axis Distance
Parent Switch = A || B || (A && C) || (B &&C)
Left Switch = A || C || (A && C) || (B && C)
Right Switch = B || C || (A && C) || (B && C)
Axis determines the dimension to calculate the axis distance from
Axis (2 bits)

Figure 28: Comparison element architecture. The comparison element does the comparisons necessary for the sorting and point propagation stages. It also assigns the new parent, left, and right center values during sorting based on custom logic computed from the A, B, and C results. Stability in this context is an indication that the condition L < P < R is satisfied. The comparison element also contains a distance module (either manhattan or euclidean), that computes the 3D distance and the axis distance.

6.5.1 Design

The comparison element is responsible for performing the comparisons necessary for the sorting and point propagation stages. For sorting, it computes the values A, B, and C which are each indicators of different violations of the condition Left < Parent < Right. The correctional outputs new left, new parent, and new right are determined based on custom logic that uses A, B, and C. A, B, and C, are also used to determine the switching commands for the left, parent, and right connections.

For point propagation, the distance module is utilized. The distance module can compute either the manhattan distance or the euclidean distance based on the user's selection. In each of these cases it provides the three-dimensional distance as well as the single-dimensional distance axis_dist. The value of axis_dist is used in two scenarios. The first is in the determination of whether a point should traverse left or right during the first traversal, and the second is in the determination of whether the second branch needs to be visited. The output signals for each of these two cases are first_direction and other_branch respectively.

6.5.2 Verification

Similar to the cluster processor, the cluster comparison element can also function independently so a testbench was written to verify its results. This testbench is cluster_ce_tb.v . This testbench includes violations of the fundamental condition between left, parent, and right (L < P < R). For all of these violations, the results for new left, new parent, and new right were all verified to be correct. Additionally, the module needed to be verified for its operation under point propagation. This involved ensuring that the point is sent in the correct direction down the tree, and that the decision of sending it to the other branch is also determined correctly and performed. On an individual modular level, this was verified to be correct and the circuit's operation was deemed to valid according to its required functionality.

6.5.3 Specifications

- 42 LUTs.
- 3 DSP blocks.

7 Hardware Models - Results

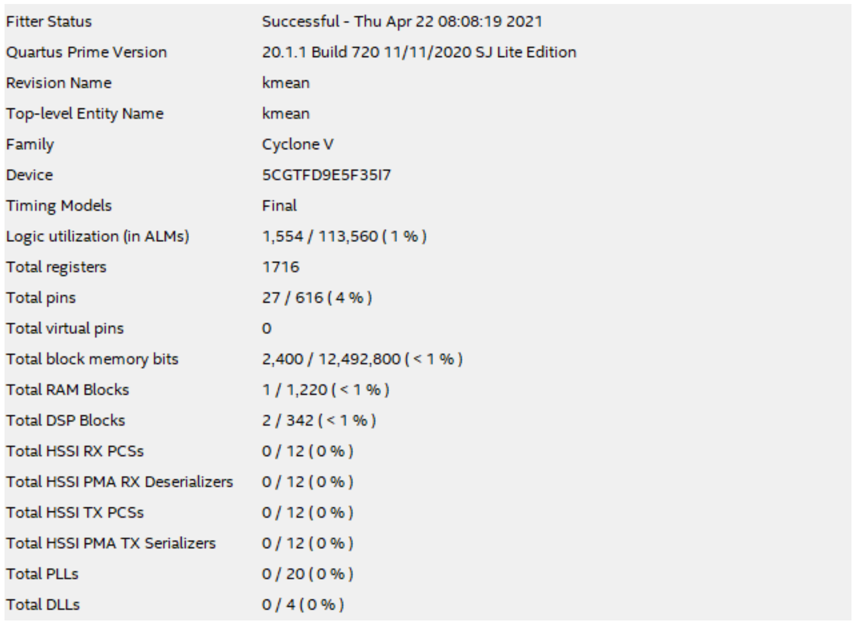
For testing the hardware performance and synthesis results we needed to test two different algorithmic improvements that we have introduced. These were division removal and kd-tree nearest cluster search. Therefore we implemented four different versions of hardware clustering circuits. The first of these was a pure sequential clustering circuit using linear nearest cluster search and with division. The second was the same sequential implementation but without division. The third was kd-tree clustering with division. Finally, the fourth was kd-tree clustering without division. The synthesis results and timing analyses for each of these is presented. Additionally, the number of cycles to cluster a specific image is also shown. Some of the hardware versions also have pipelined versions that were designed. Also note that for the kd-tree results, one node is synthesized. For all of the designed models, the distance metric used was the manhattan distance. All of the designed models were synthesized using the settings of the Cyclone V FPGA. For throughput calculations, we assume K = 1000 clusters and a value of P = 24 bits for the point size (8 bits for each dimension, 3 dimensions). The throughput calculation also assumes the binning use case.

7.1 Sequential with Divider (Reference model)

The first version that was designed was the pure sequential search model with a divider. This model was designed to be similar to how clustering algorithms run in software, and it did not have any of the optimizations that we had proposed. Therefore, here we calculate the distance metric for all of the clusters, and perform division. The search complexity in this version is $O(K)$ where K is the number of clusters. This is the baseline reference model with which all of the other improved models will be compared. Synthesis results are shown in Figure 29. The maximum frequency result in this model is **13.57 MHz**. The throughput is calculated as follows:

$$T = \frac{P * f}{2 * K} \tag{11}$$

where P is the point size, f is the maximum frequency and K is the number of clusters. The reason for the division by 2 is that we need two cycles (distance calculation, distance comparison with minimum value), to perform the necessary computations in nearest cluster determination, for a single cluster. We multiply by K because this is performed for all K clusters. Using the test parameters outlined in Section 7, we obtain a throughput value of **0.1628 Mb/s**.



Fitter Status	Successful - Thu Apr 22 08:08:19 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	kmean
Top-level Entity Name	kmean
Family	Cyclone V
Device	5CGTFD9E5F35I7
Timing Models	Final
Logic utilization (in ALMs)	1,554 / 113,560 (1 %)
Total registers	1716
Total pins	27 / 616 (4 %)
Total virtual pins	0
Total block memory bits	2,400 / 12,492,800 (< 1 %)
Total RAM Blocks	1 / 1,220 (< 1 %)
Total DSP Blocks	2 / 342 (< 1 %)
Total HSSI RX PCSs	0 / 12 (0 %)
Total HSSI PMA RX Deserializers	0 / 12 (0 %)
Total HSSI TX PCSs	0 / 12 (0 %)
Total HSSI PMA TX Serializers	0 / 12 (0 %)
Total PLLs	0 / 20 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 29: Synthesis report of the baseline sequential model with a divider.

7.2 Sequential with Pipelined Divider

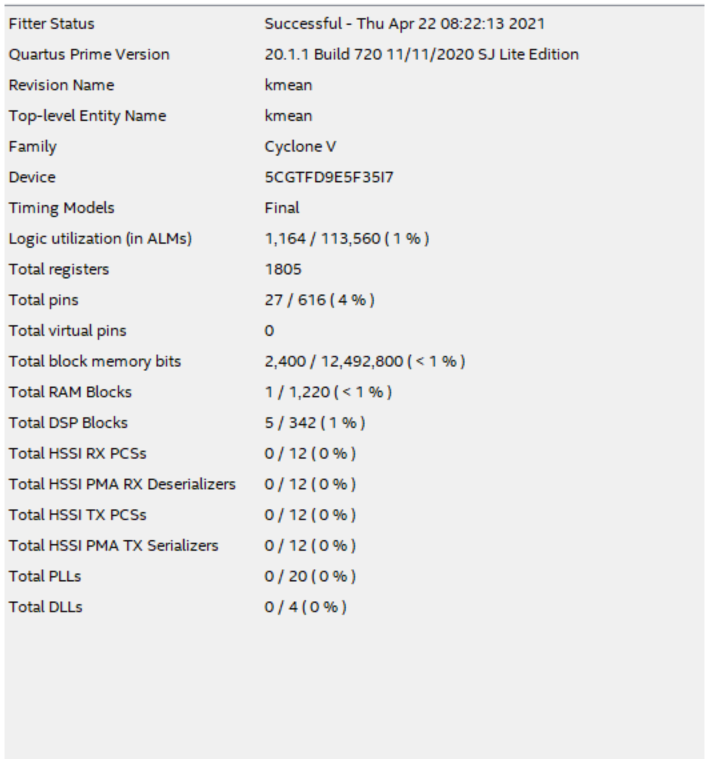
This model is almost exactly the same as the reference model but differs in the divider. The divider core configuration was changed so that it would be pipelined, allowing for 10 cycles before a result is produced. The synthesis report is the same as in Figure 29, because the change was only in the divider configuration. This gave a resultant output frequency of **65.9 MHz**. Using Equation 11 and the test parameters, the value of the throughput is **0.7908 Mb/s**.

7.3 Sequential without Divider

The improvement made in this model is the removal of division. This was done using Equation 8 for the manhattan distance, and inequality 3 for point-cluster comparison. The synthesis report for this model is shown in Figure 30. In this situation instead of dividing by two calculation stages as was done in Equation 11, we divide by three because the distance calculation is instead split into two stages. This is involves a pipeline separation of the multiplication and addition steps within the calculation of the manhattan distance. Therefore the throughput equation becomes as follows:

$$T = \frac{P * f}{3 * K} \tag{12}$$

The resultant frequency is **83.16 MHz**, and the throughput calculated using Equation 12 is **0.6653 Mb/s**.



Fitter Status	Successful - Thu Apr 22 08:22:13 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	kmean
Top-level Entity Name	kmean
Family	Cyclone V
Device	5CGTFD9E5F35I7
Timing Models	Final
Logic utilization (in ALMs)	1,164 / 113,560 (1 %)
Total registers	1805
Total pins	27 / 616 (4 %)
Total virtual pins	0
Total block memory bits	2,400 / 12,492,800 (< 1 %)
Total RAM Blocks	1 / 1,220 (< 1 %)
Total DSP Blocks	5 / 342 (1 %)
Total HSSI RX PCSs	0 / 12 (0 %)
Total HSSI PMA RX Deserializers	0 / 12 (0 %)
Total HSSI TX PCSs	0 / 12 (0 %)
Total HSSI PMA TX Serializers	0 / 12 (0 %)
Total PLLs	0 / 20 (0 %)
Total DLLs	0 / 4 (0 %)

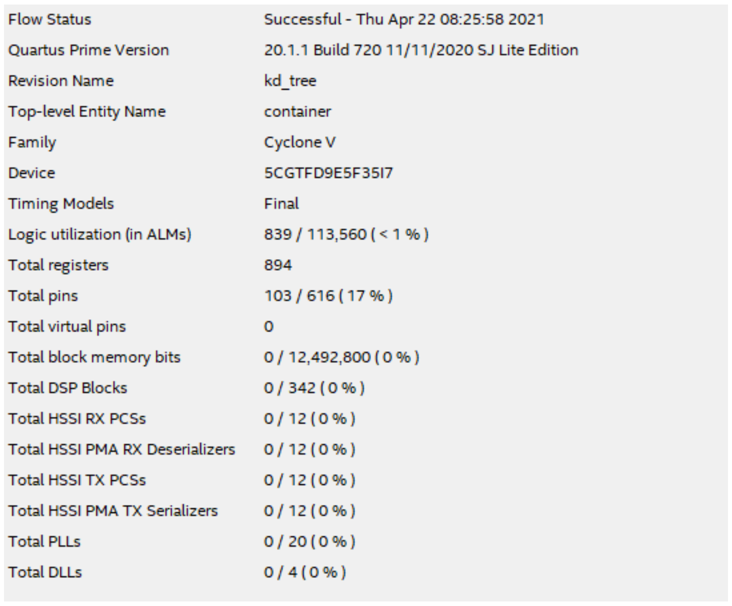
Figure 30: Synthesis report of the sequential model without division.

7.4 Three-stage Pipelined Kd Tree with Pipelined Divider

This model was the first working implementation of a Kd-tree in hardware. It was designed with the modules specified in Section 6. This version included a divider (10-stage pipelined), so that we could have a baseline comparison for division removal. The synthesis report for this version is shown in Figure 31. Pipelining was also introduced here in the comparison element. This was done in three stages due to having three main computations that occur within the comparison element. The first is the subtraction between the point and the centroid, the second is the absolute value computation, and the third is the addition of all three dimensional values. Therefore the throughput equation for this version becomes:

$$T = \frac{P * f}{2 * S * log_2(K)} \tag{13}$$

where P is the point size, f is the maximum frequency, S is the number of pipeline stages, and K is the number of clusters. Since the average case complexity of using a Kd-tree is $O(log_2(K))$, we use the log_2 operator in the equation. A division of 2 is additionally introduced because our hardware implementation requires traversal back to the root once the nearest cluster is found. The maximum frequency result for the model was **114.94 MHz**. Using Equation 13, the throughput was calculated to be **46.1339 Mb/s**.

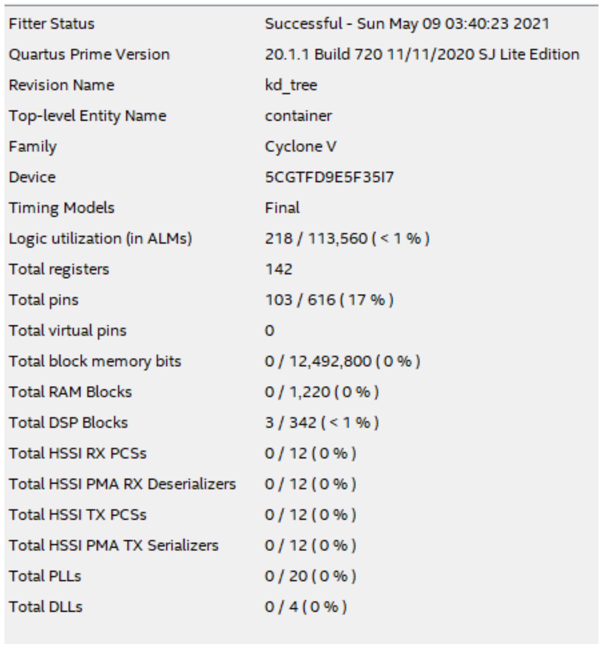


Flow Status	Successful - Thu Apr 22 08:25:58 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	kd_tree
Top-level Entity Name	container
Family	Cyclone V
Device	5CGTFD9E5F35I7
Timing Models	Final
Logic utilization (in ALMs)	839 / 113,560 (< 1 %)
Total registers	894
Total pins	103 / 616 (17 %)
Total virtual pins	0
Total block memory bits	0 / 12,492,800 (0 %)
Total DSP Blocks	0 / 342 (0 %)
Total HSSI RX PCSs	0 / 12 (0 %)
Total HSSI PMA RX Deserializers	0 / 12 (0 %)
Total HSSI TX PCSs	0 / 12 (0 %)
Total HSSI PMA TX Serializers	0 / 12 (0 %)
Total PLLs	0 / 20 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 31: Synthesis report of the initial Kd tree hardware implementation model. This model included a divider.

7.5 Three-stage Pipelined Kd Tree without Divider

This version is similar to the one that was outlined in Section 7.4, except that division was removed from it using Equations 3 and 8. The synthesis report for this version is shown in Figure 32. The maximum frequency for this model is **114.08 MHz**. This resulted in a throughput of **45.7887 Mb/s** using Equation 13.



Fitter Status	Successful - Sun May 09 03:40:23 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	kd_tree
Top-level Entity Name	container
Family	Cyclone V
Device	5CGTFD9E5F35I7
Timing Models	Final
Logic utilization (in ALMs)	218 / 113,560 (< 1 %)
Total registers	142
Total pins	103 / 616 (17 %)
Total virtual pins	0
Total block memory bits	0 / 12,492,800 (0 %)
Total RAM Blocks	0 / 1,220 (0 %)
Total DSP Blocks	3 / 342 (< 1 %)
Total HSSI RX PCSs	0 / 12 (0 %)
Total HSSI PMA RX Deserializers	0 / 12 (0 %)
Total HSSI TX PCSs	0 / 12 (0 %)
Total HSSI PMA TX Serializers	0 / 12 (0 %)
Total PLLs	0 / 20 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 32: Synthesis report for the Kd tree hardware implementation that avoids division.

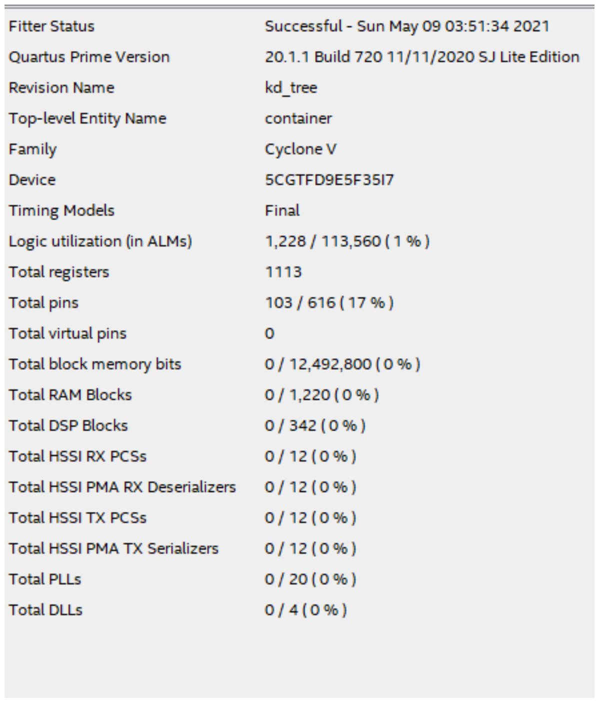
7.6 Kd-Tree with Pipelined Divider

In this Kd-tree version we include a pipelined divider but avoid pipelining the comparison element. The advantage to this is the avoidance of dividing by number of pipeline stages in the throughput equation. As a result, the throughput equation becomes as follows:

$$T = \frac{P * f}{2 * log_2(K)} \tag{14}$$

Note the absence of the S term here, in comparison to Equation 13.

The synthesis report is shown in Figure 33. The maximum frequency is **98.5 MHz**. Using this value and Equation 14, we get a throughput value of **118.6058 Mb/s**.



Fitter Status	Successful - Sun May 09 03:51:34 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	kd_tree
Top-level Entity Name	container
Family	Cyclone V
Device	5CGTFD9E5F35I7
Timing Models	Final
Logic utilization (in ALMs)	1,228 / 113,560 (1 %)
Total registers	1113
Total pins	103 / 616 (17 %)
Total virtual pins	0
Total block memory bits	0 / 12,492,800 (0 %)
Total RAM Blocks	0 / 1,220 (0 %)
Total DSP Blocks	0 / 342 (0 %)
Total HSSI RX PCSs	0 / 12 (0 %)
Total HSSI PMA RX Deserializers	0 / 12 (0 %)
Total HSSI TX PCSs	0 / 12 (0 %)
Total HSSI PMA TX Serializers	0 / 12 (0 %)
Total PLLs	0 / 20 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 33: Synthesis report for the non-pipelined Kd tree hardware implementation with a pipelined divider.

7.7 Overall Results

Design Version	**Frequency (MHz)**	**Throughput (Mb/s)**
1. Sequential with Divider	13.57	0.1628
2. Sequential with Pipelined Divider	65.90	0.7908
3. Pipelined Sequential without Divider	83.16	0.6653
4. Pipelined Kd Tree with Pipelined Divider	114.94	46.1339
5. Pipelined Kd Tree without Divider	114.08	45.7887
6. Kd Tree with Pipelined Divider	98.5	118.6058

Table 4: Frequency and throughput results for all of the designed hardware versions.

The numerous hardware versions shown above were the result of many optimization phases and continuous incremental improvements. The summary of their performance results during the binning use case is shown in Table 4. One matter of note is that all of the hardware circuits synthesized with very low resource utilization, and this can be seen in Figures 29, 30, 31, and 32.

The first improvement between the first and second version is pipelining the divider. This resulted in an almost 5x gain in the throughput. This is mainly because the divider has a high latency so pipelining it will greatly increase the frequency. The next improvement (between model 2 and 3), was to remove the division and in this case the frequency improved but it did not result in enough of an improvement in the throughout during binning. This is because of adding an extra multiplication step during binning. Whether or not an improvement can be observed during clustering is something that must further be investigated.

The next set of versions (4, 5, and 6), were based on using a Kd tree and it is clear that this has provided a tremendous boost in the throughput. This is because of dividing by a $log_2(K)$ term instead of K, which is because of the cluster filtering during tree traversal. When comparing models 2 and 4, we observe an approximate throughput gain of 58. As for the comparison between models 4 and 5, we find that during binning there is almost no improvement in the throughput when avoiding division. This will most likely be observed during clustering, which is where the divider will be used for every iteration. The most interesting observation is the consequence of the removal of the pipelining stages between models 4 and 6. The point of interest is that while model 6 understandably had a lower frequency, the resultant throughput was higher. This indicates that the higher frequency in model 4 was not enough to compensate for the additional number of cycles prevalent due to pipelining the comparison element. If we also remove the divider in model 6, the frequency and throughput results will most likely not change significantly, as was observed in the difference in results between models 4 and 5.

Therefore we can conclude that for binning, model 6 was the most optimal out of all the design versions. As was previously stated, further testing and experimentation will be required for the clustering use case. This will be more complicated due to the variable nature of clustering, however this can be overcome by taking average cases, looking deeper into the nature of the input data, and observing the effects of varying the hyperparameters. Once a solid approach is formulated for testing clustering, more concrete and final conclusions can be drawn. It must be noted that the 40 Gb/s ethernet requirement cannot be achieved at this stage, however this design can be scaled for future work, to support multiple parallel trees processing simultaneously. This will allow the throughput to be multiplied by the number of active processing trees which will in turn allow for the achievement of a throughput of 40 Gb/s.

8 Conclusion

This senior design project report has addressed some of the problems associated with clustering in hardware, and has proposed a novel hardware design as a solution. The hardware design was based on the kd-tree approach which is a known efficient method for cluster filtration. New algorithmic improvements were introduced through this design, which were primarily the avoidance of division, the independent parallel sorting strategy, as well as the sub-box initialization method. These improvements were based off of extensive theoretical work and iterative verification through the utilization of a golden model that was implemented in software. The project targets were accomplished, as the results showed that the requirements were met within the physical hardware constraints.

Despite the project's accomplishments, further improvements can be made through future work and experimentation. One of these is to find a way detect the shape of the data distribution so that clusters are not initialized outside of it. Solving this problem will ensure that all clusters will be assigned points and the empty cluster problem will be avoided. Furthermore, an actual network component needs to be designed so that the system use cases can be tested practically. This will require using cloud vendor services so that the clustering can be done for real data. One important aspect is also to test and verify the design's operation on an actual FPGA. This will give us a more accurate demonstration of how the system will function in a practical scenario. Finally, the parallel tree processing strategy needs to be implemented so that the throughput is multiplied enough times to meet the 40 Gb/s ethernet requirement.

Overall, many lessons were learned through iterative design and experimentation. Work on this design was highly rewarding, especially when the practical results from theoretical improvements were observed. We are satisfied with what we have accomplished in this project and hope to continue improving on it to realize its manifestation as an actual commercial project.

References

[1] Altera. lpm_divide megafunction user guide, Jun 2007.

[2] A.D. Booth and A.J.T. Colin. On the efficiency of a new method of dictionary construction. *Information and Control*, 3(4):327–334, 1960.

[3] Susan Craw. *Manhattan Distance*, pages 639–639. Springer US, Boston, MA, 2010.

[4] Youbo Hou, Dan Xiong, Tonglin Jiang, Lili Song, and Qi Wang. Social media addiction: Its impact, mediation, and intervention. *Cyberpsychology: Journal of Psychosocial Research on Cyberspace*, 13, 02 2019.

[5] Hanaa Hussain, Khaled Benkrid, Huseyin Seker, and Ahmet Erdogan. Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 248 – 255, 07 2011.

[6] Plotly Technologies Inc. Collaborative data science, 2015.

[7] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

[8] James B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[10] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[11] T. Saegusa and T. Maruyama. An fpga implementation of k-means clustering for color images based on kd-tree. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6, 2006.

[12] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. *CoRR*, abs/1906.02243, 2019.

[13] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. Fpga-based k-means clustering using tree-based data structures. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–6, 2013.

Appendix

All code associated with this project, including the commit history is found on the github repository at: https://github.com/hsfalsharif/KMeansClusteringSW . The github also includes the individual contributions of each member (hsfalsharif is Hamza, and nitrogar is Ahmad). The file tree.py is the golden model. All of files and directories that are of importance to what we have worked on are outlined in the README of the repository. All verilog code is found in the 'verilog' directory.