MSc Thesis

Implementation of Hardware Accelerators for Computer Vision Algorithms with VHDL

Christoforos-Arhythios Glias

Supervisor: Spyridon Nikolaidis, Professor
Tuesday 1st October, 2019
Abstract

Hardware acceleration is the use of hardware specially designed to perform specific functions more efficiently than software running on a general-purpose CPU. Some of the advantages of hardware against software include speedup, lower power consumption, lower latency and increased parallelism, at the cost of longer development times and reduced ability to update the designs after manufacturing.

In this work, a hardware accelerator was designed for the Pointcloud Library (PCL) which is used for computer vision applications. Most of the algorithms inside PCL have the same computations as their core. A hardware unit was designed for these core computations using VHDL and synthesized using Xilinx’s tool Vivado. This unit could be used to perform the core computations alongside software implementations, significantly speeding up the execution of the algorithm, allowing the user to increase the payload of the algorithm and achieve better, real-time results.
Περίληψη

Hardware acceleration είναι η χρήση ειδικού υλικού, σχεδιασμένο αποκλειστικά για τον υπολογισμό συγκεκριμένων διεργασιών με καλύτερη απόδοση από λογισμικό που θα τρέχει σε γενικού σκοπού επεξεργαστή. Κάποια από τα πλεονεκτήματα της χρήσης υλικού αντί για λογισμικό για τον υπολογισμό διεργασιών είναι: η αύξηση ταχύτητας των υπολογισμών, η μείωση της κατανάλωσης ενέργειας, η μικρότερη καθυστέρηση μεταξύ διαδοχικών υπολογισμών και η αύξηση της χρήσης παραλληλισμού του συστήματος, κόστος μεγαλύτερο χρόνο σχεδιασμού και παραγωγής καθώς και μειωμένη δυνατότητα αναβάθμισης του συστήματος μετά την παραγωγή.

Στην παρούσα εργασία σχεδιάστηκε υλικό επιτάχυνσης για την βιβλιοθήκη αλγορίθμων Pointcloud Library (PCL), η οποία χρησιμοποιείται για εφαρμογές Τεχνητής Όρασης (Computer Vision). Σχεδόν όλοι οι διαφορετικοί αλγόριθμοι της εν λόγω βιβλιοθήκης χρησιμοποιούν τους ίδιους υπολογισμούς στον πυρήνα τους. Για αυτόν τον πυρήνα σχεδιάστηκε ειδικό υλικό με χρήση της γλώσσας VHDL το οποίο συνθέθηκε χρησιμοποιώντας το εργαλείο της Xilinx, Vivado. Αυτό το υλικό μπορεί να χρησιμοποιηθεί παράλληλα με λογισμικό ώστε να επιτευχθεί σημαντική αύξηση της ταχύτητας υπολογισμού των αλγορίθμων επιτρέποντας έτσι την χρήση τον όγκο των δεδομένων που επεξεργάζεται και να πετύχει καλύτερα αποτελέσματα σε πραγματικό χρόνο.
# Table of Contents

List of Figures

<table>
<thead>
<tr>
<th>List of Figures</th>
</tr>
</thead>
<tbody>
<tr>
<td>ix</td>
</tr>
</tbody>
</table>

1 3D Point Feature Representations

<table>
<thead>
<tr>
<th>1.1 Surface Normals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1 Theoretical Primer</td>
</tr>
<tr>
<td>1.1.2 Calculating the scale</td>
</tr>
<tr>
<td>1.1.3 Calculating Surface Normals</td>
</tr>
<tr>
<td>1.2 Point Feature Histograms (PFH)</td>
</tr>
<tr>
<td>1.2.1 Theoretical Primer</td>
</tr>
<tr>
<td>1.2.2 Calculating PFH</td>
</tr>
<tr>
<td>1.3 Fast Point Feature Histograms (FPFH)</td>
</tr>
<tr>
<td>1.3.1 Theoretical Primer</td>
</tr>
<tr>
<td>1.3.2 Computing FPFH</td>
</tr>
<tr>
<td>1.4 View Feature Histograms (VFH)</td>
</tr>
<tr>
<td>1.4.1 Theoretical Primer</td>
</tr>
<tr>
<td>1.4.2 Calculating VFH</td>
</tr>
</tbody>
</table>

2 Pair Features Algorithm

| 2.1 Overview |
| 2.2 Modifications |
| 2.2.1 Divisions |
| 2.2.2 Inverse Cosine |
| 2.2.3 Pipeline |
| 2.3 Final |

3 Modules

| 3.1 Xilinx Primitives |
| 3.1.1 Floating-point Adder/Subtractor |
| 3.1.2 Floating-point Multiplier |
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.3</td>
<td>Floating-point Multiplier-Adder</td>
<td>25</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Delay</td>
<td>25</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Floating-Point Divider</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Floating-Point Adder</td>
<td>27</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Overview</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Pseudocode</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Floating-Point Multiplier</td>
<td>29</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Overview</td>
<td>29</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Pseudocode</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Delay</td>
<td>30</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Overview</td>
<td>30</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Pseudocode</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>Dot Product</td>
<td>30</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Overview</td>
<td>30</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Pseudocode</td>
<td>32</td>
</tr>
<tr>
<td>3.6</td>
<td>Cross Product</td>
<td>33</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Overview</td>
<td>33</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Pseudocode</td>
<td>35</td>
</tr>
<tr>
<td>3.7</td>
<td>Reciprocal Square Root</td>
<td>36</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Overview</td>
<td>36</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Pseudocode</td>
<td>39</td>
</tr>
<tr>
<td>3.8</td>
<td>Euclidean Distance</td>
<td>40</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Overview</td>
<td>40</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Pseudocode</td>
<td>41</td>
</tr>
<tr>
<td>3.9</td>
<td>Vector Magnitude</td>
<td>42</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Overview</td>
<td>42</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Pseudocode</td>
<td>43</td>
</tr>
<tr>
<td>3.10</td>
<td>Inverse Tangent</td>
<td>44</td>
</tr>
<tr>
<td>3.10.1</td>
<td>Overview</td>
<td>44</td>
</tr>
<tr>
<td>3.10.2</td>
<td>Pseudocode</td>
<td>47</td>
</tr>
<tr>
<td>3.11</td>
<td>Pair Features</td>
<td>48</td>
</tr>
<tr>
<td>3.11.1</td>
<td>Overview</td>
<td>48</td>
</tr>
<tr>
<td>3.11.2</td>
<td>Pseudocode</td>
<td>51</td>
</tr>
<tr>
<td>3.12</td>
<td>Final Implementation</td>
<td>52</td>
</tr>
<tr>
<td>3.12.1</td>
<td>Resources</td>
<td>52</td>
</tr>
<tr>
<td>3.12.2</td>
<td>Simulation</td>
<td>52</td>
</tr>
</tbody>
</table>

A Floating Point Arithmetic | 55
A.1 Representation | 55
# TABLE OF CONTENTS

A.2 Addition - Subtraction ........................................... 56
A.3 Multiplication ..................................................... 57

B Codes ........................................................................... 59
B.1 Floating Point Subadder .......................................... 59
B.2 Floating Point Multiplier ......................................... 62
B.3 Dot Product .......................................................... 64
B.4 Cross Product ....................................................... 66
  B.4.1 Cross Element .................................................. 66
  B.4.2 Cross Product .................................................. 67
B.5 Reciprocal Square Root .......................................... 68
  B.5.1 Newton - Raphson Element ............................... 68
  B.5.2 RSR) ............................................................. 70
B.6 Euclidean Distance ............................................... 71
B.7 Normal Vector ...................................................... 73
B.8 Atan ....................................................................... 74
  B.8.1 Single Input Atan ............................................. 74
  B.8.2 Two Input Atan ............................................... 77
B.9 Pair Features ......................................................... 80
B.10 Testbench ........................................................... 93
B.11 C functions ........................................................ 98

Bibliography ..................................................................... 107
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Surface Normals estimations before orientation correction. 2</td>
</tr>
<tr>
<td>1.2</td>
<td>Surface Normals estimations after re-orienting. 3</td>
</tr>
<tr>
<td>1.3</td>
<td>Differences between smaller and bigger scales in surface normals estimation 4</td>
</tr>
<tr>
<td>1.4</td>
<td>Interconnections in the $k$-neighborhood for PFH. 5</td>
</tr>
<tr>
<td>1.5</td>
<td>The $uvw$ frame of reference. 6</td>
</tr>
<tr>
<td>1.6</td>
<td>Examples of PFH for two different points in a cloud. 7</td>
</tr>
<tr>
<td>1.7</td>
<td>Interconnections in the $k$-neighborhood for FPFH. 9</td>
</tr>
<tr>
<td>1.8</td>
<td>Example FPFH histogram. 10</td>
</tr>
<tr>
<td>1.9</td>
<td>Interconnections in the $k$-neighborhood for FPFH. 12</td>
</tr>
<tr>
<td>1.10</td>
<td>Interconnections in the $k$-neighborhood for FPFH. 13</td>
</tr>
<tr>
<td>1.11</td>
<td>Interconnections in the $k$-neighborhood for FPFH. 13</td>
</tr>
<tr>
<td>2.1</td>
<td>The Pair Features Algorithm’s flowchart. 16</td>
</tr>
<tr>
<td>2.2</td>
<td>Plot of the inverse cosine function. 18</td>
</tr>
<tr>
<td>2.3</td>
<td>The Pair Features Algorithm’s modified flowchart. 21</td>
</tr>
<tr>
<td>3.1</td>
<td>Block diagramm of the Dot Product calculation module. 32</td>
</tr>
<tr>
<td>3.2</td>
<td>Cross product vector in a right-handed coordinate system. 33</td>
</tr>
<tr>
<td>3.3</td>
<td>Block diagramm of the Cross Element calculation module. 34</td>
</tr>
<tr>
<td>3.4</td>
<td>Block diagramm of the Cross Product calculation module. 35</td>
</tr>
<tr>
<td>3.5</td>
<td>Block diagramm of the Newton-Raphson calculation module. 38</td>
</tr>
<tr>
<td>3.6</td>
<td>Block diagramm of the Reciprocal Square Root module. 38</td>
</tr>
<tr>
<td>3.7</td>
<td>Block diagramm of the Reciprocal Euclidean Distance calculation module. 41</td>
</tr>
<tr>
<td>3.8</td>
<td>Block diagramm of the Reciprocal Vector Magnitude calculation module. 43</td>
</tr>
<tr>
<td>3.9</td>
<td>The approximation polynomial (doted line) compared with the arctan function (solid line). 45</td>
</tr>
<tr>
<td>3.10</td>
<td>Block diagramm of the arctangent calculation module. 46</td>
</tr>
<tr>
<td>3.11</td>
<td>Block diagramm of the two-arguments arctangent calculation module. 47</td>
</tr>
<tr>
<td>3.12</td>
<td>Block diagramm of the final Pair Features calculation module. 50</td>
</tr>
</tbody>
</table>
3.13 Simulation results for the Pair Features module. .......................... 53
3D Point Feature Representations

As point feature representations go, surface normals and curvature estimates are somewhat basic in their representations of the geometry around a specific point. Though extremely fast and easy to compute, they cannot capture too much detail, as they approximate the geometry of a point’s k-neighborhood with only a few values. As a direct consequence, most scenes will contain many points with the same or very similar feature values, thus reducing their informative characteristics.

This section introduces a family of 3D feature descriptors coined PFH (Point Feature Histograms) for simplicity, presents their theoretical advantages and discusses implementation details from PCL’s perspective.

There are three main algorithms to compute these PFH descriptors and they will each be discussed in detail.

1.1 Surface Normals

1.1.1 Theoretical Primer

Surface Normals are important properties of a geometric surface and are used in many areas like computer graphics.

If the geometric surface is known, then it is easy to calculate the direction of the normal vector since it is perpendicular to the surface at that point. In our case though we do not know the geometric surface, but what we do know is a set of points of the surface. Considering this, we have two options.

- to calculate the actual surface from the set of points, through surface meshing, and then calculate the normal vector
- to approximate the normal vector from the set of points.

In this work, the second option is being used. There are various different methods on how to calculate the normal vectors, but we will be focusing on the simplest.
If we consider the task of calculating the normal vector on a point of the surface to be equivalent with calculating the normal on an tangential plane at that point, the task is reduced to least squares plane fitting estimation problem.

That way, to calculate the normal all that needs doing is to analyze the eigen-vectors and eigen-values of a covariance array (PCA -Principal Component Analysis). For every point \( p_i \) the covariance array \( C \) is created, using the formula:

\[
C = \frac{1}{k} \sum_{i=1}^{k} (p_i - \bar{p}) \cdot (p_i - \bar{p})^T
\]

\[
C \cdot \vec{v}_j = \lambda_j \cdot \vec{v}_j, \quad j \in \{0, 1, 2\}
\]

where \( k \) is the size of the \( k \)-neighborhood of \( p_i \), \( \bar{p} \) is the 3D center of the nearest points, \( \lambda_j \) is the \( j^{th} \) eigenvalue of the covariance array, and \( \vec{v}_j \) is the \( j^{th} \) eigenvector of the array.

Figure 1.1: Surface Normals estimations before orientation correction.

Due to the unavailability of a clear mathematical way to calculate the polarity of the normal using the above method, the orientation of the vectors is not consistent though all the points. This is shown in Figure 1.1 in two sections of a larger set of points. The right part of the figure presents the Extended Gaussian Image (EGI), also known as the normal sphere, which describes the orientation of all normals from the point cloud.

Solving this issue is simple if the viewpoint \( v_p \) (the point from where the data where captured), is known to us. To correctly re-orient all the normals \( \vec{n}_i \) towards \( v_p \), they just have to satisfy the following equation.

\[
\vec{n}_i \cdot (v_p - p_i) > 0
\]
1.1. SURFACE NORMALS

Figure 1.2 presents the result of re-orienting all the normals of Figure 1.1.

![Surface Normals estimations after re-orienting.](image)

**Figure 1.2**: Surface Normals estimations after re-orienting.

1.1.2 Calculating the scale

It was already mentioned that the normal of a surface must be calculated from the points in the $k$-neighborhood of the point.

This raised another issue, since the most appropriate number $k$ of neighbors or radius $r$ to use has to be determined.

This can not be formalized and automated, so the user has to pick a number according to the requirements of the application. In Figure 1.3 we can see the potential differences between a small scale (small $k$ or $r$) versus a bigger one. In the left part of the figure, the selection of scale factor is reasonably well chosen. The normals are approximately perpendicular to the two planar surfaces and small edges across the table are visible.

In contrast, in the right part of the figure, because of the bigger scale used, the set of neighbors is covering points from adjacent surfaces, which distorts the estimations. Normals are rotated at the edges of the planar surfaces, edges get smeared and finer details are supressed.

So, the user must take care to select a scale that is small enough to capture the level of detail required by the application.
CHAPTER 1. 3D POINT FEATURE REPRESENTATIONS

1.1.3 Calculating Surface Normals

The algorithm to calculate the surface normals is:

for every point $p$ in the set of points $P$

1. find the nearest neighbors of $p$.
2. compute the surface normal $n$ of $p$.
3. check if $n$ is correctly oriented towards the viewpoint $v_p$ and flip otherwise.

Figure 1.3: Differences between smaller and bigger scales in surface normals estimation
1.2 Point Feature Histograms (PFH)

1.2.1 Theoretical Primer

The goal of the PFH formulation is to encode a point’s $k$-neighborhood geometrical properties by generalizing the mean curvature around the point using a multi-dimensional histogram of values. This highly dimensional hyperspace provides an informative signature for the feature representation, is invariant to the 6D pose of the underlying surface, and copes very well with different sampling densities or noise levels present in the neighborhood.

A Point Feature Histogram representation is based on the relationships between the points in the $k$-neighborhood and their estimated surface normals. Simply put, it attempts to capture as best as possible the sampled surface variations by taking into account all the interactions between the directions of the estimated normals. The resultant hyperspace is thus dependent on the quality of the surface normal estimations at each point.

![Interconnections in the $k$-neighborhood for PFH.](image)

Figure 1.4 presents an influence region diagram of the PFH computation for a query point ($p_q$), marked with red and placed in the middle of a circle (sphere in 3D) with radius $r$, and all its $k$ neighbors (points with distances smaller than the radius $r$) are fully interconnected in a mesh. The final PFH descriptor is computed as a histogram of relationships between all pairs of points in the neighborhood, and thus has a computational complexity of $O(k^2)$.

To compute the relative difference between two points $p_i$ and $p_j$ and their associated normals $n_i$ and $n_j$, a fixed coordinate frame at one of the points is defined (Figure 1.5).
\[ \mathbf{u} = \mathbf{n}_i \]
\[ \mathbf{v} = (p_j - p_i) \times \mathbf{u} \]
\[ \mathbf{w} = \mathbf{u} \times \mathbf{v} \]

Figure 1.5: The \textit{uvw} frame of reference.

Using the above \textit{uvw} frame, the difference between the two normals \( n_s \) and \( n_t \) can be expressed as a set of angular features as follows:

\[ \alpha = \mathbf{v} \cdot \mathbf{n}_t \]
\[ \phi = \mathbf{u} \cdot \frac{(p_t - p_s)}{d} \]
\[ \theta = \arctan(\mathbf{w} \cdot \mathbf{n}_t, \mathbf{u} \cdot \mathbf{n}_t) \]

where \( d \) is the Euclidean distance between the two points.

\[ d = \|p_t - p_s\|_2 \]

The quadruplet \([\alpha, \phi, \theta, d] \) is computed for each pair of two points in \( k \)-neighborhood, therefore reducing the 12 values (\( xyz \) and normal information) of the two points and their normals to 4.

To create the final PFH representation for the query point, the set of all quadruplets is binned into a histogram. The binning process divides each features’s value range into \( b \) subdivisions, and counts the number of occurrences in each subinterval. Since 3 out of the 4 features presented above are measure of the angles between normals, their values can easily be normalized to the same interval on the trigonometric circle. A binning example is to divide each feature interval into the same number of equal parts, and therefore create a histogram with \( b^4 \) bins in a fully correlated space. In this space, a histogram bin increment corresponds
to a point having certain values for all its 4 features. Figure 1.6 presents examples of Point Feature Histograms representations for different points in a cloud.

![Figure 1.6: Examples of PFH for two different points in a cloud.](image)

In some cases, the fourth feature, $d$, does not present an extreme significance for 2.5D datasets, usually acquired in robotics, as the distance between neighboring points increases from the viewpoint. Therefore, omitting $d$ for scans where the local point density influences this feature dimension has proved to be beneficial.

### 1.2.2 Calculating PFH

The algorithm to calculate the PHE computation is:

for each point $p$ in cloud $P$

1. get the nearest neighbors of $p$
2. for each pair of neighbors, compute the 3 angular values $(\alpha, \phi, \theta,)$ (optionally the distance $d$)
3. bin all the results in an output histogram.

where cloud $P$ is the input point cloud that contains the points.
1.3 Fast Point Feature Histograms (FPFH)

1.3.1 Theoretical Primer

The theoretical computational complexity of the Point Feature Histogram (PFH) for a given point cloud $P$ with $n$ points is $O(nk^2)$, where $k$ is the number of neighbors for each point $p$ in $P$. For real-time or near real-time applications, the computation of Point Feature Histograms in dense point neighborhoods can represent one of the major bottlenecks.

A simplification of the PFH formulation is called Fast Point Feature Histograms (FPFH), that reduces the computational complexity of the algorithm to $O(nk)$, while still retaining most of the discriminative power of the PFH.

- In a first step, for each query point $p_q$ a set of tuples $(\alpha, \phi, \theta)$ between itself and its neighbors are computed as described in Point Feature Histograms (PFH) descriptors - this will be called the Simplified Point Feature Histogram (SPFH);
- In a second step, for each point its $k$ neighbors are re-determined, and the neighboring SPFH values are used to weight the final histogram of $p_q$ (called FPFH) as follows:

$$FPFH(p_q) = SPFH(p_q) + \frac{1}{k} \sum_{i=1}^{k} \frac{1}{\omega_k} \cdot SPFH(p_k)$$

where the weight $\omega_k$ represents a distance between the query point $p_q$ and a neighbor point $p_k$ in some given metric space, thus scoring the $(p_q, p_k)$ pair, but could just as well be selected as a different measure if necessary. To understand the importance of this weighting scheme, Figure 1.7 presents the influence region diagram for a $k$-neighborhood set centered at $p_q$.

Thus, for a given query point $p_q$, the algorithm first estimates its SPFH values by creating pairs between itself and its neighbors (illustrated using red lines). This is repeated for all the points in the dataset, followed by a re-weighting of the SPFH values of $p_q$ using the SPFH values of its $p_k$ neighbors, thus creating the FPFH for $p_q$. The extra FPFH connections, resultant due to the additional weighting scheme, are shown with black lines. As the diagram shows, some of the value pairs will be counted twice (marked with thicker lines in the figure).
1.3. FAST POINT FEATURE HISTOGRAMS (FPFH)

Figure 1.7: Interconnections in the $k$-neighborhood for FPFH.

Differences between PFH and FPFH

The main differences between the PFH and FPFH formulations are summarized below:

1. the FPFH does not fully interconnect all neighbors of $p_q$ as it can be seen from Figure 1.7, and is thus missing some value pairs which might contribute to capture the geometry around the query point
2. the PFH models a precisely determined surface around the query point, while the FPFH includes additional point pairs outside the $r$ radius sphere (though at most $2r$ away)
3. because of the re-weighting scheme, the FPFH combines SPFH values and recaptures some of the point neighboring value pairs
4. the overall complexity of FPFH is greatly reduced, thus making possible to use it in real-time applications
5. the resultant histogram is simplified by decorrelating the values, that is simply creating $d$ separate feature histograms, one for each feature dimension, and concatenate them together (see Figure ).
1.3.2 Computing FPFH

The algorithm to compute FPFH is:

for each point \( p \) in cloud \( P \)

1. first pass:
   i. get the nearest neighbors of \( p \)
   ii. or each pair of \( p_q - p_k \) compute the three angular values \( (\alpha, \phi, \theta) \)
   iii. bin all the results in an output SPFH histogram

2. second pass:
   i. get the nearest neighbors of \( p_q \)
   ii. use each SPFH of every neighboring point \( p_k \) with a weighting scheme to assemble the FPFH of \( p_q \).

```
k // desired size of neighborhoods
Q <- {} // queue of pts with uncompleted neighborhoods
// queue entries also hold a list \( n_{qi} \) of \( k \) neighbors and a \( sid_{qi} \) entry – see below
for all scanline \( S = s_i \) do
  increase(\( sid \)) // current scan line id number
  P ← {} // list of points ready for processing
  for all points \( q_j \in Q \) do
    for all points \( s_i \in S \) do
      if \( s_i \) is closer to \( q_i \) than \( q_i \)'s current neighbors then
        \( n_{qi} \leftarrow n_{qi} \cup s_i \)
      if \( |n_{qi}| > k \) then
        delete most distant point from \( n_{qi} \)
        update \( sid_{qi}, sid \)
      if \( sid_{qj} < sid \) then
        // \( q_i \)'s neighborhood unaffected by cur. scan line
        \( P \leftarrow P \cup q_j \)
        \( Q \leftarrow Q \backslash q_j \)
  for all points \( s_i \in S \) do
```

Figure 1.8: Example FPFH histogram.
1.3. FAST POINT FEATURE HISTOGRAMS (FPFH)

\[ Q \leftarrow Q \cup s_j \]

Initialize \( n_{s_i} \) by using the same algorithm backwards

\[ // \text{ go back through last scanlines until } s_i \text{ stays unchanged} \]

for all points \( p_i \in P \) do

compute FPFH on \( p_i \) 's neighborhood \( n_{p_i} \)
1.4 View Feature Histograms (VFH)

1.4.1 Theoretical Primer

The Viewpoint Feature Histogram (or VFH) has its roots in the FPFH descriptor. FPFH is leveraged due to its speed, discriminative power and strong recognition results, but viewpoint variance is also added in, while retaining invariance to scale.

VFH’s contribution to the problem of object recognition and pose identification comes from extending the FPFH to be estimated for the entire object cluster (as seen in the Figure 1.9), and computing additional statistics between the viewpoint direction and the normals estimated at each point. To do this, the key idea of mixing the viewpoint direction directly into the relative normal angle calculation in the FPFH is used.

![Figure 1.9: Interconnections in the k-neighborhood for FPFH.](image)

The viewpoint component is computed by collecting a histogram of the angles that the viewpoint direction makes with each normal. Note, not the view angle to each normal as this would not be scale invariant, but instead the angle between the central viewpoint direction translated to each normal.

\[ \theta = \arccos \left( n_p \cdot \frac{v - p}{\|v - p\|_2} \right) \]

The second component measures the relative pan, tilt and yaw angles \( \alpha, \phi, \theta \) as described in Fast Point Feature Histograms (FPFH) descriptors but now measured between the viewpoint direction at the central point and each of the normals on the surface.
1.4. VIEW FEATURE HISTOGRAMS (VFH)

The new assembled feature is therefore called the Viewpoint Feature Histogram (VFH). Figure 1.11 presents this idea with the new feature consisting of two parts:

1. a viewpoint direction component and
2. a surface shape component comprised of an extended FPFH.

The major difference between the PFH/FPFH descriptors and VFH, is that for a given point cloud dataset, only a single VFH descriptor will be estimated, while the resultant PFH/FPFH data will have the same number of entries as the number of points in the cloud.
1.4.2 Calculating VFH

The algorithm to calculate the VFH computation is:

1. Calculate the centroid $p_c$ and its normal $n_c$.
2. For every point $p_q$ in the cloud $P$ calculate the angle between their normal $n_q$ and $n_c$.
3. Bin the results in the viewpoint component.
4. Calculate FPFH at $p_c$ considering the entire point cloud $P$ as it’s neighbours.
5. Add the two histograms together.

, where cloud $P$ is the input point cloud that contains the points, centroid $p_c$ is the point that results from averaging the $xyz$ coordinates of all points and $n_c$ is the normalized vector between the viewpoint and the centroid.
In the core of all three algorithms that were described in Chapter 2, (and all other PCL feature descriptor algorithms that were not mentioned because they are based on those three), lies the calculation of the uwv and the quadruplet $(\alpha, \phi, \theta, d)$.

Since this is the basic and also the most computationally expensive part of all of them and it is repeated many times (once for every pair of points), it makes a prime candidate for a hardware accelerator.

In this chapter we take a deeper look inside the algorithm used to compute the frame and quadruplet of features, hereafter called the Pair Features Algorithm, as well as some modifications that were made to it in order to make it more hardware friendly.

### 2.1 Overview

As has already been shown in chapter 1.2.1, to compute the Pair Features for two points $p_i$ and $p_j$ first we the Derboux frame is calculated at one of them.

This frame (Figure 1.5) is calculated by the following equations:

\[
\begin{align*}
\mathbf{u} &= \mathbf{n}_i \\
\mathbf{v} &= (p_j - p_i) \times \mathbf{u} \\
\mathbf{w} &= \mathbf{u} \times \mathbf{v}
\end{align*}
\]

where:

- $p_j - p_i = (p_{jx} - p_{ix}, p_{jy} - p_{iy}, p_{jz} - p_{iz})$
- $\mathbf{n}_i, \mathbf{n}_j$ are the corresponding surface normals on $p_i, p_j$

After the frame has been calculated, we calculate the $(\alpha, \phi, \theta, d)$ angular values with:

\[
\begin{align*}
\alpha &= \mathbf{v} \cdot \mathbf{n}_t \\
\phi &= \mathbf{u} \cdot \frac{(p_i - p_s)}{d} \\
\theta &= \arctan(\mathbf{w} \cdot \mathbf{n}_t, \mathbf{u} \cdot \mathbf{n}_t)
\end{align*}
\]
where $d$ is the Euclidean distance between the two points.

Figure 2.1: The Pair Features Algorithm’s flowchart.

Figure 2.1 presents the algorithm’s flowchart as is implemented in the PCL library.

The algorithm takes as inputs the two points in cartesian coordinates $(p_x, p_y, p_z)$ and the two normals, also in cartesian coordinates $(n_x, n_y, n_z)$ and outputs the 3 features $(\alpha, \phi, \theta)$.

Unfortunately, the algorithm is completely sequential and so we can not introduce concurrency to accelerate it. This means that our only option is to use a deeply pipelined design.
2.2 Modifications

There are a couple modifications that can be done on the algorithm to make it more hardware friendly. One is to avoid divisions and the other one is to avoid the calculation of the inverse cosine function (acos).

2.2.1 Divisions

Division in hardware is very slow and often difficult to implement. Therefore, when it is possible it is preferable to multiply by the reciprocal number instead.

\[
\frac{a}{\hat{\beta}} = a \frac{1}{\hat{\beta}}
\]

There are two divisions taking place in our algorithm. Once when we divide with the Euclidean distance \(d\) to calculate \(\theta_1\) and \(\theta_2\) and once more when we normalize vector \(\mathbf{v}\) by dividing it with its magnitude.

\[
d = \sqrt{(p_{jx} - p_{ix})^2 + (p_{jy} - p_{iy})^2 + (p_{jz} - p_{iz})^2}
\]

\[
|\mathbf{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}
\]

As we can see from the formulas above, in both cases we are essentially dividing with the square root of a number. That means that if we calculate the reciprocal square root of the number, we can avoid all divisions in the algorithm and use multiplications instead.

It is also worth mentioning that if the square root of the number is needed, we can still calculate it by multiplication after we have computed the reciprocal square root, since:

\[
\sqrt{x} = x \frac{1}{\sqrt{x}}
\]

2.2.2 Inverse Cosine

While the Derboux frame is being calculated, there is a chance that the polarity of the \(\mathbf{v}\) vector would be towards the inside of the surface, depending on the point that was selected as a base. Since the points are taken from a 2.5D surface, all vectors should point towards the viewpoint, away from the surface.

The solution to this issue is to use the other point as a base when this happens. To make sure the right point is always picked as a base, the point whose normal forms the smaller angle with the line that connects the two points is selected.
CHAPTER 2. PAIR FEATURES ALGORITHM

As is seen from the flow chart this is done by using the inverse cosine function (acos) to calculate the angles.

But, since the acos function is always decreasing, as shown in Figure 2.2 that means that calculation of acos can be avoided, simply because:

\[
\arccos(|\theta_1|) \geq \arccos(|\theta_2|) \iff |\theta_2| \leq |\theta_1|
\]

![Figure 2.2: Plot of the inverse cosine function.](image)

2.2.3 Pipeline

It was already mentioned that the best optimization that can be done hardware wise to the algorithm is to use pipeline to increase the throughput of data.

But the original flow of the algorithm has 2 conditions to end the execution of the algorithm early (\(d = 0\) or \(|v| = 0\)). This means that the algorithm has variable length of execution which is not compatible with a pipelined flow.

The reason for these early end conditions is to avoid division by zero. But, since division was replaced with multiplication, we can neglect them.
That way the algorithm will have a constant length and can be pipelined. At the end of the algorithm though, there should be a check of whether the original end conditions would have occurred and set the features to zero, so that this version will be consistent with the original.
CHAPTER 2. PAIR FEATURES ALGORITHM

2.3 Final

The final version of the algorithm after the modifications can be seen in Figure 2.3.

To test if there is any significant improvement in the modified version of the algorithm’s performance, the algorithm was run and measured on a single pointcloud 30 times to simulate a 30 fps input camera. The results are presented in the following array.

<table>
<thead>
<tr>
<th></th>
<th>Original Version</th>
<th>Modified Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min (ns)</td>
<td>1275</td>
<td>1233</td>
</tr>
<tr>
<td>Max (ns)</td>
<td>24829</td>
<td>23388</td>
</tr>
<tr>
<td>Avg (ns)</td>
<td>1545.06</td>
<td>1473.31</td>
</tr>
<tr>
<td>Total (ms)</td>
<td>64.1511</td>
<td>61.1718</td>
</tr>
</tbody>
</table>

As we can see, an overall improvement of approximately 3ms or 5% was achieved. These results do not take into account the pipeline that will be used in the hardware implementation which will add even more time efficiency.
Figure 2.3: The Pair Features Algorithm’s modified flowchart.
After analyzing and modifying the algorithm in the previous chapter we can now go into details about the modules used to implement it. All these modules were developed in VHDL hardware description language and were simulated using Xilinx’s Vivado Suite.

Here is a list of all the sub-modules needed to implement the algorithm:

- Floating-point adder
- Floating-point Multiplier
- Dot product calculation
- Cross product calculation
- Reciprocal square root calculation
- Euclidean distance calculation
- Vector magnitude calculation

3.1 Xilinx Primitives

Xilinx provides some IPs that can be used when you develop a module on their FPGA. Those IPs are parametric and fully optimized for their platforms. It is usually a good idea to use them where applicable.

Although custom logic was developed for every sub-module, and it is presented later on in the chapter, the primitives provided by Xilinx where used in the end for better results in synthesis and implementation.

3.1.1 Floating-point Adder/Subtractor

A floating-point adder that takes two vectors representing single precision numbers and outputs their sum after 8 clocks latency.

Formula:

\[ result = A + B \]

Inputs:
CHAPTER 3. MODULES

**The clock used.**

- `s_axis_a_tdata` Number A
- `s_axis_a_tvalid` Flag that indicates number A is valid
- `s_axis_b_tdata` Number B
- `s_axis_b_tvalid` Flag that indicates number B is valid

Outputs:

- `m_axis_result_tdata` resulting number, sum of the inputs
- `m_axis_result_tvalid` Flag that indicates the result is valid

The same primitive exists in subtraction mode where the formula is:

\[
result = A - B
\]

### 3.1.2 Floating-point Multiplier

A floating-point multiplier that takes two vectors representing single precision numbers and outputs their product with 8 clocks latency.

Formula:

\[
result = A \times B
\]

Inputs:

- `aclk` The clock used.
- `s_axis_a_tdata` Number A
- `s_axis_a_tvalid` Flag that indicates number A is valid
- `s_axis_b_tdata` Number B
- `s_axis_b_tvalid` Flag that indicates number B is valid

Outputs:

- `m_axis_result_tdata` resulting number, product of the inputs
- `m_axis_result_tvalid` Flag that indicates the result is valid
3.1.3 Floating-point Multiplier-Adder

A floating-point multiplier that takes three vectors representing single precision numbers and outputs the result of the following formula with a latency of 16 clocks).

Formula:

\[ \text{result} = (A \times B) + C \]

Inputs:

- `aclk` The clock used.
- `s_axis_a_tdata` Number A
- `s_axis_a_tvalid` Flag that indicates number A is valid
- `s_axis_b_tdata` Number B
- `s_axis_b_tvalid` Flag that indicates number B is valid
- `s_axis_c_tdata` Number C
- `s_axis_c_tvalid` Flag that indicates number C is valid

Outputs:

- `m_axis_result_tdata` Resulting number, product of the inputs
- `m_axis_result_tvalid` Flag that indicates the result is valid

The same primitive exists in subtraction mode where the formula is:

\[ \text{result} = (A \times B) - C \]

3.1.4 Delay

A simple delay IP that takes an input and outputs it after a specified number of clocks.

Formula:

\[ D(t) = Q(t - n) \]

where \( n \) is the number of clock periods we want to delay by.

Inputs:

- `clk` The clock used.
- `D` Input vector we want delayed

Outputs:

- `Q` Delayed value of D
3.1.5 Floating-Point Divider

An IP that takes as input two floating-point numbers and outputs the result of their division. It is pipelined and has a latency of 28 clocks.

Formula:
\[ \text{result} = \frac{A}{B} \]

Inputs:

- **aclk** The clock used.
- **s_axis_a_tdata** Number A
- **s_axis_a_tvalid** Flag that indicates number A is valid
- **s_axis_b_tdata** Number B
- **s_axis_b_tvalid** Flag that indicates number B is valid

Outputs:

- **m_axis_result_tdata** resulting number, product of the inputs
- **m_axis_result_tvalid** Flag that indicates the result is valid
3.2 Floating-Point Adder

3.2.1 Overview

A floating-point adder that takes two vectors representing single precision numbers and outputs their sum on the next clock (latency 1).

Formula:

\[ \text{result} = A + B \]

Inputs:

- \( \text{clk} \) The clock used.
- \( a \) Number A
- \( b \) Number B

Outputs:

- \( \text{result} \) resulting number, sum of the inputs

For subtraction you just need to reverse the sign bit of one of the two inputs since:

\[ A - B = A + (-B) \]
\[ B - A = (-A) + B \]

3.2.2 Pseudocode

```c
// Break inputs to their components
a_sign, a_exponent, a_mantissa = a[31], a[30:23], a[22:0]
b_sign, b_exponent, b_mantissa = b[31], b[30:23], b[22:0]

// Special Cases
if a_exponent == "0x00" // 0 +/- X = X
    result = b
else if b_exponent == "0x00" // X +/- 0 = X
    result = a
else if (a == NaN) || (b == NaN) // X + NaN
    result = NaN
else if (a[30:0] == inf) && (b[30:0] == inf)
    if a_sign != b_sign // inf - inf = NaN
        result = NaN
    else // +/-inf +/- inf = +/-inf
        result = a
else if a == inf // +/-inf + X = +/-inf
    result = a
else if b == inf // X +/- inf = +/-inf
    result = b
```
else // Normal Case
    if a_exponent > b_exponent // Scale stage
        scale b
        result_exponent = a_exponent
    else if b_exponent > a_exponent
        scale a
        result_exponent = b_exponent
    if a_sign = b_sign // A+B
        result_mantissa = a_mantissa + b_mantissa
        result_sign = a_sign
    else if a_mantissa >= b_mantissa // A-B
        result_mantissa = a_mantissa - b_mantissa
        result_sign = a_sign
    else // B-A
        result_mantissa = b_mantissa - a_mantissa
        result_sign = b_sign
    result = (result_sign, result_exponent, result_mantissa)
    normalize result

return result
3.3 Floating-Point Multiplier

3.3.1 Overview

A floating-point multiplier that takes two vectors representing single precision numbers and outputs their product on the next clock (latency 1).

Formula:

\[ \text{result} = A \times B \]

Inputs:

- \( \text{clk} \): The clock used.
- \( \text{a} \): Number A
- \( \text{b} \): Number B

Outputs:

- \( \text{result} \): Resulting number, product of the inputs

3.3.2 Pseudocode

```c
// Break inputs to their components
a_sign , a_exponent , a_mantissa = a[31] , a[30:23] , a[22:0]
b_sign , b_exponent , b_mantissa = b[31] , b[30:23] , b[22:0]

// Special Cases
if (a == NaN) || (b == NaN)    // X * NaN = NaN
    result = NaN
else if (a == inf) || (b == inf)
    if (a == 0) || (b == 0)   // 0 * inf = NaN
        result = NaN
    else
        result = inf
        result_sign = a_sign xor b_sign
else
    // Normal Cases
    result_exponent = a_exponent + b_exponent
    result_mantissa = a_mantissa + b_mantissa
    result = (result_sign, result_exponent, result_mantissa)
    round result
    normalize result

return result
```

29
3.4 Delay

3.4.1 Overview

A simple delay IP that takes an input and outputs it after a specified number of clocks.

Formula:

\[ D(t) = Q(t - n) \]

where \( n \) is the number of clock periods we want to delay by.

Inputs:

- \texttt{clk} The clock used.
- \texttt{D} Input vector we want delayed

Outputs:

- \texttt{Q} Delayed value of \texttt{D}

3.4.2 Pseudocode

```python
reg = [N]  # N clks delay
@rising_edge(clk)
    reg[0] = D  # input
    for (i = N; i==1; i--)
        reg[1] = reg[i-1]
Q = reg[31]  # output
return Q
```

3.5 Dot Product

3.5.1 Overview

The dot product, or inner product, is an algebraic operation that takes two coordinate vector and returns a single scalar number.

Algebraically the dot product of two vectors \( \mathbf{a} = [a_1, a_2, ..., a_n] \) and \( \mathbf{b} = [b_1, b_2, ..., b_n] \) is defined as:

\[
\mathbf{a} \cdot \mathbf{b} = \sum_{n=1}^{N} a_i b_i
\]

\[= a_1 b_1 + a_2 b_2 + ... + a_n b_n\]

In our case we are working with 3D vectors so \( N = 3 \) and the formula ends up as:

\[
\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + a_3 b_3
\]
3.5. DOT PRODUCT

The dot product module two sets of coordinates as inputs representing the two vectors to be multiplied, and outputs the result which is a single number.

Inputs:

- **clk**  The clock used.
- **vector1x**  The x coordinate of the first vector
- **vector1y**  The y coordinate of the first vector
- **vector1z**  The z coordinate of the first vector
- **vector2x**  The x coordinate of the second vector
- **vector2y**  The y coordinate of the second vector
- **vector2z**  The y coordinate of the second vector

Outputs:

- **result**  Dot product of the vectors

Submodules:

- 1× floating-point multiplier
- 2× floating-point multiplier and adder

Figure 3.1 shows the block diagram of the implementation.
### 3.5.2 Pseudocode

```c
// Inputs a[3] and b[3]
// vectors' coordinates in array form
result = 0
for (i=0; i<3; i++)
    result += a[i]*b[i]

return result
```

![Figure 3.1: Block diagram of the Dot Product calculation module.](image-url)
3.6 Cross Product

3.6.1 Overview

The cross product, or vector product, is an operation on two vectors in 3D space. Given two linearly independent vectors \( \mathbf{a} \) and \( \mathbf{b} \) the cross product \( \mathbf{a} \times \mathbf{b} \) is a vector perpendicular to both of them, as shown in Figure 3.2.

In coordinate notation each vector \( \mathbf{a} \) and \( \mathbf{b} \) can be defined as the sum of 3 orthogonal components parallel to the standard basis vectors.

\[
\mathbf{a} = a_1 \mathbf{i} + a_2 \mathbf{j} + a_3 \mathbf{k} \quad \mathbf{b} = b_1 \mathbf{i} + b_2 \mathbf{j} + b_3 \mathbf{k}
\]

Then, the cross product \( \mathbf{a} \times \mathbf{b} \) can be expanded as:

\[
\mathbf{a} \times \mathbf{b} = (a_1 \mathbf{i} + a_2 \mathbf{j} + a_3 \mathbf{k}) \times (b_1 \mathbf{i} + b_2 \mathbf{j} + b_3 \mathbf{k})
\]

\[
= a_1 b_1 (\mathbf{i} \times \mathbf{i}) + a_1 b_2 (\mathbf{i} \times \mathbf{j}) + a_1 b_3 (\mathbf{i} \times \mathbf{k}) +
\]

\[
a_2 b_1 (\mathbf{j} \times \mathbf{i}) + a_2 b_2 (\mathbf{j} \times \mathbf{j}) + a_2 b_3 (\mathbf{j} \times \mathbf{k}) +
\]

\[
a_3 b_1 (\mathbf{k} \times \mathbf{i}) + a_3 b_2 (\mathbf{k} \times \mathbf{j}) + a_3 b_3 (\mathbf{k} \times \mathbf{k})
\]

Since the three basis vectors are all orthogonal with each other, the cross product of two of them equals the third one. With that in mind, as well as that the cross product of a vector with itself equals zero, the formula can be further simplified to:

\[
\mathbf{a} \times \mathbf{b} = a_1 b_1 \mathbf{i} + a_2 b_2 \mathbf{j} + a_3 b_3 \mathbf{k} + a_2 b_1 \mathbf{j} + a_3 b_2 \mathbf{k} + a_3 b_1 \mathbf{k} - a_2 b_2 \mathbf{i} - a_3 b_3 \mathbf{j} - a_2 b_1 \mathbf{k}
\]

\[
= (a_2 b_3 - a_3 b_2) \mathbf{i} + (a_3 b_1 - a_1 b_3) \mathbf{j} + (a_1 b_2 - a_2 b_1) \mathbf{k}
\]

meaning that we can consider the vector to be of the form \( \mathbf{a} \times \mathbf{b} = s_1 \mathbf{i} + s_2 \mathbf{j} + s_3 \mathbf{k} \) where the three scalar components are:

\[
s_1 = a_2 b_3 - a_3 b_2
\]

\[
s_2 = a_3 b_1 - a_1 b_3
\]

\[
s_3 = a_1 b_2 - a_2 b_1
\]

It is easy to see that the scalar components have the same format, so a submodule was designed that would calculate them called cross element. It takes as input the four coordinates and outputs the scalar component. It’s block diagram is shown in Figure 3.3.

**Figure 3.2**: Cross product vector in a right-handed coordinate system

| Inputs: |
| clk | The clock used. |

33
\textbf{vec11} The first coordinate of the first vector
\textbf{vec12} The second coordinate of the first vector
\textbf{vec21} The first coordinate of the second vector
\textbf{vec22} The second coordinate of the second vector

\textbf{Outputs:}
\textbf{result} The scalar component

\textbf{Submodules:}
\begin{itemize}
  \item 1× floating-point multiplier
  \item 2× floating-point multiplier and subtractor
\end{itemize}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.3}
\caption{Block diagram of the Cross Element calculation module.}
\end{figure}

The cross product module takes as input 2 sets of coordinates representing the two vectors to be multiplied and outputs one set of coordinates for the product vector. Because each output is independent from the others, three cross elements can be used in parallel to speed the operation. Figure 3.4 shows the block diagram.

\textbf{Inputs:}
\textbf{clk} The clock used.
\textbf{vec1x} The x coordinate of the first vector
\textbf{vec1y} The y coordinate of the first vector
\textbf{vec1z} The z coordinate of the first vector
\textbf{vec2x} The x coordinate of the second vector
\textbf{vec2y} The y coordinate of the second vector
\textbf{vec2z} The z coordinate of the second vector

34
3.6. CROSS PRODUCT

Outputs:

resultx The scalar component for the x-axis
resulty The scalar component for the y-axis
resultz The scalar component for the z-axis

Submodules:

- 3× cross element

![Block diagram of the Cross Product calculation module.](image)

Figure 3.4: Block diagramm of the Cross Product calculation module.

3.6.2 Pseudocode

```plaintext
function cross_element(v11, v12, v21, v22)
    elem = v11*v21 - v12*v22
    return elem

// Inputs v1[3], v2[3]
// coordinates in array form
result_x = cross_element(v1[1], v2[2], v1[2], v2[1])
result_y = cross_element(v1[2], v2[0], v1[0], v2[2])
result_z = cross_element(v1[0], v2[1], v1[1], v2[0])
```
3.7 Reciprocal Square Root

3.7.1 Overview

To approximate the reciprocal square root function we are using the Fast Inverse Square Root algorithm.

The algorithm accepts a single precision floating-point number $x$, approximates the logarithm $\log_2 x$ as an initial estimate for the $\frac{1}{\sqrt{x}}$ and then uses the Newton-Raphson method to refine the result.

Initial estimate:
We are using a logarithm to approximate the inverse square root, because:

$$\log_2\left(\frac{1}{\sqrt{x}}\right) = -\frac{1}{2} \log_2(x)$$

Since $x$ is a positive floating-point number, it can be written as:

$$x = (1 + m_x)2^e$$

where $m_x, e_x$ are the mantissa and exponent of $x$.

Then

$$\log_2(x) = e_x + \log_2(1 + m_x) \approx e_x + m_x$$

since

$$m_x \epsilon [0, 1) \rightarrow \log(1 + m_x) \approx m_x$$

If we interpret the floating-point number as an integer we get:

$$I_x = E_x L + M_x$$

$$= L(e_x + B + m_x)$$

$$\approx L \log_2(x) + LB$$

where $E_x = e_x + B$ is the biased exponent, $B = 127$, $M_x = m_x \times L$ and $L = 2^{23}$. If we use this interpretation in the original equation we get:

$$\log_2\left(\frac{1}{\sqrt{x}}\right) = -\frac{1}{2} \log_2(x)$$

$$\frac{I_y}{L} - B \approx -\frac{1}{2} \left(\frac{I_x}{L} - b\right)$$

$$I_y \approx \frac{3}{2} LB - \frac{1}{2} I_x$$

The first term, called the magic number, is approximated by $\frac{3}{2}LB = 0x5F375A86$ which has been found to work best, while the second term can be calculated by simple right shift on $x$. 
Newton-Raphson method:
The next step is to use the Newton-Raphson method to refine the approximation. It is a numerical analysis root-finding method that states: If there is an approximation $y_n$ for $y$, a better approximation can be calculated by $y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$.

For the inverse square root this works out as:

$$y = x^{-2}$$
$$y^2 = \frac{1}{x}$$

$\rightarrow f(y) = y^2 - x = 0$

so

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$$
$$= y_n - \frac{y_n^2 - x}{-2y_n^3}$$
$$= y_n + \frac{y_n^3(y_n^{-2} - x)}{2}$$
$$= y_n(3 - \frac{x}{y_n^2})$$

The absolute error after 2 Newton-Raphson iterations is $\Delta_{max} \approx 3.684 \cdot 10^{-6}$ [Moroz] which is more than enough for our application.

The Newton-Raphson method was implemented in a separate submodule since it was needed two times. Figure 3.5 presents the block diagramm. It has a latency of 4 clks.

**Inputs:**

- **clk** The clock used.
- **init_estimate** The initial estimation to be refined
- **input_number** The input number of the original function

**Outputs:**

- **resul** The refined approximation

**Submodules:**

- 3× floating-point multiplier
- 1× floating-point subtractor

37
The reciprocal square root module’s block diagram can be seen in Figure 3.6. It is worth noting that with an extra Newton-Raphson iteration the result would be bit accurate for single precision floating-point numbers.

**Inputs:**

- **clk**  The clock used.
- **input_number** The input number.

**Outputs:**

- **result** The final approximation

**Submodules:**

- 1× floating-point subtractor
- 3× Newton-Raphson modules
3.7.2 Pseudocode

```plaintext
function newton_raphson(y, x_0)
    y = y*(3-x_0*y*y)/2
    return y

// Input number x
y0 = 0x5f375a86- (x>>1)  // gives initial guess y0
y1 = newton_raphson(y0, x)  // Newton step 1st iteration
y2 = newton_raphson(y1, x)  // Newton step 2nd iteration
return y2  // Final approximation
```
CHAPTER 3. MODULES

3.8 Euclidean Distance

3.8.1 Overview

As has already been discussed in Chapter 2.2, instead of the normal Euclidean Distance we are using the Reciprocal Euclidean Distance. The module to calculate that takes as inputs two sets of coordinates of the points in 3D space, and outputs the reciprocal of the distance between them.

The formula is:

$$
\frac{1}{d} = \frac{1}{\sqrt{(p_{jx} - p_{ix})^2 + (p_{jy} - p_{iy})^2 + (p_{jz} - p_{iz})^2}}
$$

If we define a new vector \( \mathbf{v} \) as \( \mathbf{v} = \mathbf{p}_2 - \mathbf{p}_1 = (p_{2x} - p_{1x}, p_{2y} - p_{1y}, p_{2z} - p_{1z}) \) we can then rewrite the formula as:

$$
\frac{1}{d} = \frac{1}{\sqrt{\mathbf{v} \cdot \mathbf{v}}}
$$

which means we can easily implement it by using the Dot Product and Reciprocal Square Root module. The block diagram is presented in Figure 3.6. The module has a latency of 14 clks.

Inputs:

- **clk**  The clock used.
- **p1_x**  The x coordinate of the first point
- **p1_y**  The y coordinate of the first point
- **p1_z**  The z coordinate of the first point
- **p2_x**  The x coordinate of the second point
- **p2_y**  The y coordinate of the second point
- **p2_z**  The z coordinate of the second point

Outputs:

- **result**  The reciprocal distance between the two points

Submodules:

- 3× floating-point subtractor
- 1× dot product calculation module
- 1× reciprocal square root calculation module
3.8. EUCLIDEAN DISTANCE

3.8.2 Pseudocode

```c
// Coordinates of input points
// in array form
p1[3], p2[3]
for (i=0;i<3;i++)
    v[i] = p2[i] - p1[i] // intermediate vector
temp = dot_product(v, v) // v * v
result = rsr(temp) // 1/sqrt(x)
return result
```

Figure 3.7: Block diagram of the Reciprocal Euclidean Distance calculation module.
3.9 Vector Magnitude

3.9.1 Overview

In accordance with the modifications that were made in in Chapter 2.2, instead of the normal vector magnitude we are using it’s reciprocal. The module to calculate that takes as input one sets of coordinates for the vector in 3D space, and outputs the reciprocal of it’s magnitude. This module is very similar to the reciprocal Euclidean distance.

The formula is:

\[
\frac{1}{|\mathbf{v}|} = \frac{1}{\sqrt{v_x^2 + v_y^2 + v_z^2}}
\]

which again, we can rewrite as:

\[
\frac{1}{|\mathbf{v}|} = \frac{1}{\sqrt{\mathbf{v} \cdot \mathbf{v}}}
\]

so that we can use the Dot Product and Reciprocal Square Root modules in the implementation. The block diagramm is presented in Figure 3.8.

Inputs:

- clk  The clock used.
- v_x  The x coordinate of the first point
- v_y  The y coordinate of the first point
- v_z  The z coordinate of the first point

Outputs:

- resul The reciprocal distance between the two points

Submodules:

- 1× dot product calculation module
- 1× reciprocal square root calculation module
3.9. VECTORMAGNITUDE

Figure 3.8: Block diagram of the Reciprocal Vector Magnitude calculation module.

### 3.9.2 Pseudocode

// Coordinates of input vector
// in array form
v[3]
temp = dot_product(v, v) // v * v
result = rsr(temp) // 1/sqrt(x)
return result
3.10 Inverse Tangent

3.10.1 Overview

The inverse tangent, or arctangent, is defined as the angle in the Euclidean plane, in radians, between the positive x-axis and the vector to the point \((x, y) \neq (0, 0)\).

The final part of the algorithm requires the implementation of the two-argument arctangent function, or \(\arctan 2\). The difference between the two-argument and single-argument arctangent is that the single-argument function can not distinguish between diametrically opposite directions.

E.g. The arctangent between the x-axis and \(v_1 = (1, 1)\) is \(\frac{\pi}{4}\), which is the same as the arctangent between the x-axis and \(v_2 = (-1, -1)\) because \(\frac{1}{1} = \frac{-1}{1}\).

The \(\arctan 2\) function calculates one unique arctan value from the inputs and then uses their signs to determine the quadrant of the result and selects the correct branch of the arctangent.

E.g. \(\arctan 2(1, 1) = \frac{\pi}{4}\) while \(\arctan 2(-1, -1) = -\frac{3\pi}{4}\).

The way it determines the desired result is:

\[
\arctan 2(y, x) = \begin{cases} 
\arctan \left( \frac{y}{x} \right), & \text{if } x > 0 \\
\arctan \left( \frac{y}{x} \right) + \frac{\pi}{2}, & \text{if } x < 0 \text{ and } y \geq 0 \\
\arctan \left( \frac{y}{x} \right) - \frac{\pi}{2}, & \text{if } x < 0 \text{ and } y < 0 \\
\frac{\pi}{2}, & \text{if } x = 0 \text{ and } y > 0 \\
-\frac{\pi}{2}, & \text{if } x = 0 \text{ and } y < 0 \\
\text{undefined}, & \text{if } x = 0 \text{ and } y = 0 
\end{cases}
\]

To approximate the \(\arctan\) function we used an optimal 3-term polynomial.

\[
\arctan(x) = Ax^5 + Bx^3 + Cx
\]

, where

\[
A = 0.0776509570923569 \\
B = -0.287434475393028 \\
C = \frac{\pi}{4} - A - B
\]

The maximum error of this approximation has been found through brute force for all values \(\frac{\pi}{n}\) for \(n \in [-100, 100]\) to be 0.00084283 rads or 0.004826%.

Figure 3.9 shows the plots of the approximation and the proper \(\arctan\) function. It is obvious that within the bounds \([-1,1]\) which we care about, since the two inputs to the function are cosines, the two plots are an almost absolute fit.
The arctan submodule takes as input one single precision floating-point number \( x \) and outputs \( \arctan(x) \). It is fully pipelined with latency 8 and throughput 1 result per clk. Figure 3.10 shows the block diagram.

Inputs:

- \textbf{clk}  The clock used.
- \textbf{x}  The input number

Outputs:

- \textbf{result}  The arctan of the input

Submodules:

- \( 2 \times \) floating-point adders
- \( 4 \times \) floating-point multipliers
The arctan2 module takes as input one single precision floating-point number \( x \) and outputs \( \text{arctan2}(x) \). It uses the arctan from above, plus some logic to determine the quadrant. Figure 3.10 shows the block diagram. Note that since this module has custom logic, this diagram should not be considered an accurate representation of how the RTL would be synthesized. The delay modules are not depicted for simplicity.

**Inputs:**

- **clk** The clock used.
- **x** The input number

**Outputs:**

- **result** The arctan of the input

**Submodules:**

- 2× floating-point adders
- 1× floating-point divider
- 1× single argument arctan calculation module
Figure 3.11: Block diagram of the two-arguments arctangent calculation module.

### 3.10.2 Pseudocode

```c
#define PI 3.1415926535897932384626433832795
#define PI_2 (PI/2.0)
#define PI_4 (PI/4.0)
#define A 0.0776509570923569
#define B -0.287434475393028
#define C (PI_4 - A - B)

if x == 0
    // x == 0
    if y > 0
        result = PI_2
    else if y < 0
        result = -PI_2
    else
        result == NaN
else
    z = y/x
    atan = ((A*z*z + B)*z*z + C)*z
    if x > 0
        result = atan
    else
        // x < 0
        if y >= 0
            result = atan + PI
        else
            result = atan - PI

return result
```


3.11 Pair Features

3.11.1 Overview

The final module is the one that computes the Pair Features Algorithm itself. It makes use of all the modules that have been discussed previously in this Chapter. It takes 4 sets of 3D coordinates as inputs, 2 for the two points, and 2 for their respective normal vectors. It is fully pipelined with a latency of 83 clks and throughput 1 result per clk. Figure 3.12 presents the block diagram.

Inputs:

- **clk** The clock used.
- **p1_x** The x coordinate of the first point
- **p1_y** The y coordinate of the first point
- **p1_z** The z coordinate of the first point
- **p2_x** The x coordinate of the second point
- **p2_y** The y coordinate of the second point
- **p2_z** The z coordinate of the second point
- **n1_x** The x coordinate of the first normal
- **n1_y** The y coordinate of the first normal
- **n1_z** The z coordinate of the first normal
- **n2_x** The x coordinate of the second normal
- **n2_y** The y coordinate of the second normal
- **n2_z** The z coordinate of the second normal

Outputs:

- **feature0** The first feature of the two points
- **feature1** The second feature of the two points
- **feature2** The third feature of the two points
- **feature3** The fourth feature of the two points

Submodules:

- 6\times THETA clk delay
- 6\times DP2P1 clk delay
- 1\times RSR0 clk delay
- 1\times V0 clk delay
- 1\times FEATURE1 clk delay

48
• 1× FEATURE2 clk delay
• 1× FEATURE3 clk delay
• 3× V clk delay
• 1× ATANINY clk delay
• 6× N0 clk delay
• 3× N1 clk delay
• 3× N2 clk delay
• 3× N21 clk delay
• 3× floating-point subtractor
• 5× floating-point multipliers
• 5× dot product calculation module
• 2× cross product calculation module
• 1× reciprocal vector magnitude calculation module
• 1× reciprocal Euclidean distance calculation module
• 1× Atan2 calculation module
Figure 3.12: Block diagram of the final Pair Features calculation module.
3.11.2 Pseudocode

// Coordinate inputs for points
// and normal vectors
// in array form
p1[3], p2[1], n1[3], n2[3]

f[3] = reciprocal_distance(p1, p2)
for (i = 0; i < 3; i++)
    dp2pi = p2[i] - p1[i]

angle1 = dot_product(n1, dp2p1) * f[3]  // angle between n1 and points line
angle2 = dot_product(n2, dp2p1) * f[3]
if (abs(angle2) > abs(angle1))  // compare absolute values of angles
    swap(n1, n2)
    f[2] = -angle2
    for (i = 0; i < 3; i++)
        dp2pi *= -1
else
    f[2] = angle1

v = cross_product(dp2p1, n1)
v_magnitude = reciprocal_magnitude(v)
for (i = 0; i < 3; i++)
    v[i] *= v_magnitude  // normalize vector v

w = cross_product(n1, v)
f[1] = dot_product(v, n2)
temp1 = dot_product(v, n2)
temp2 = dot_product(n1, n2)
f[0] = atan2(temp1, temp2)
if (v_magnitude == inf) || (f[3] == inf)  // check for invalid conditions
    for (i = 0; i < 3; i++)
        f[i] = 0  // set all features to 0

return f
3.12 Final Implementation

3.12.1 Resources

The module was synthesized using Xilinx’s Vivado Suite.

Using the report generated by this tool we saw that the resources used to synthesize the whole module were:

<table>
<thead>
<tr>
<th>Slice LUTs</th>
<th>Slice Registers</th>
<th>F7 Muxes</th>
<th>F8 Muxes</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>19676</td>
<td>132943</td>
<td>278</td>
<td>68</td>
<td>116</td>
</tr>
</tbody>
</table>

We can see that the most heavily used resource were the Slice Registers (Flip-Flops), which makes sense if we consider that our design is fully pipelined.

The post-synthesis simulation results are discussed in the next section.

3.12.2 Simulation

Although the resulting module has a high latency of 365 clocks latency because it is fully pipelined, a throughput of 1 clock was achieved. This means that after the first result is calculated, it will only take 1 clock for the next result to be outputed. Provided that the stream of inputs remains uninterrupted, this provides huge efficiency.

The same Pointcloud file that was used to run the software implementation was used as input to the module for consistency, with the only modification being that the numbers were converted to Hex format.

The cloud consists of 538 points. Taking into account the neighbourhood relationship between those points for the FPGA algorithm, it results into 1383 combinations of pairs.

A simple testbench was developed in VHDL were the input files were read from text files and they were pass on to the module as needed. For the simulation, Xilinx’s Vivado Simulator was used.

The clock used in the simulation had 4ns period or else 250MHz frequency. Taking into account the latency and throughput of the module, this means that it would take 1460ns until the first output is computed, and then a new one would be outputed every 4ns, needing 5532ns for all the pairs to be calculated.

The Comparison between the hardware module and the software implementation running on the same pointcloud 30 times to simulate a 30fps input video, is shown in the following table:
As we can see, an acceleration of $\approx 382$ times was achieved comparing to the original SW implementation, and $\approx 365$ times comparing to the modified one.

Figure 3.13 shows the simulation results. Unfortunately the signals looked squashed as the simulation window had to be zoomed-out a lot in order to capture all 1383 pair features calculations in the screen.

The first time marker at $2156\text{ns}$ marks the beginning of the input stream of data. Second marker at $3616\text{ns}$ marks the output of the first result after 365 clocks, and the final marker at $9148\text{ns}$ the last result.
Floating Point Arithmetic

Floating point arithmetic (FP) is arithmetic using approximation of real numbers so as to support a trade-off between range and precision. A floating-point system can be used to represent, with a fixed number of digits, numbers of different orders of magnitude: e.g. distance in an Astronomical or Atomic scale. Because of this dynamic range the numbers that can be represented are not uniformly spaced; the difference between two consecutive representable numbers grows with the chosen scale.

There are many floating-point representations, but the most commonly used one is the IEE 754 Standard for Floating-Point Arithmetic which is the one used by modern day computers.

A.1 Representation

A number that can be represented exactly by floating-point representation is of the form:

\[ \text{significant} \times \text{base}^{\text{exponent}} \]

where significant, base and exponent are all integers. The significant is also called mantissa.

For example:

\[ 1.2345 = \underbrace{12345}_{\text{significant}} \times \underbrace{10}_{\text{base}}^{-4} \]

Single precision floating-point numbers are 32 bits longs. They are packed in a datum as sign bit (1–bit), exponent field (8–bits) and the significant or mantissa (23), from left to right.

The exponent in binary formats, instead of being stored as positive or negative it is stored as an unsigned number with a bias of 127. Also, the leading 1 of the normalized significant is not actually stored in the datum but it is implicit.
There are also Denormal numbers. Those are numbers that to be represented in a normal way would require an exponent below the smallest representable exponent. Those numbers are represented with a mantissa with leading zeros instead and the exponent equal to zero.

The special values of the representation are:

- $+0$ $s = 0, \text{exp} = 0, \text{mantissa} = 0$
- $-0$ $s = 1, \text{exp} = 0, \text{mantissa} = 0$
- $+\infty$ $s = 0, \text{exp} = 255, \text{mantissa} = 0$
- $-\infty$ $s = 1, \text{exp} = 255, \text{mantissa} = 0$
- $\text{NaN}$ $s = \pm 1, \text{exp} = 255, \text{mantissa} \neq 0$

### A.2 Addition - Subtraction

Addition in floating-point arithmetic is calculated by first representing both numbers using the same exponent. Then it is just a matter of adding the mantissas together.

For example, considering the following numbers:

$$123456.7 = 1.234567 \times 10^5$$
$$101.7654 = 1.017654 \times 10^2$$

The calculation goes like this:

$$123456.7 + 101.7654 = (1.234567 \times 10^5) + (1.017654 \times 10^2)$$
$$= (1.234567 \times 10^5) + (0.001017654 \times 10^5)$$
$$= (1.234567 + 0.001017654) \times 10^5$$
$$= 1.235584654 \times 10^5$$

The result is then rounded and normalized if needed.

Subtraction is completely equivalent.

Special operations:

- $x \pm 0 = x$
- $\pm 0 \pm 0 = +0$
- $x \pm \infty = \pm \infty$
- $\pm 0 \pm \infty = \pm \infty$
- $+\infty - \infty = \text{NaN}$
A.3 Multiplication

To multiply, the significands are multiplied while the exponents are added, and the result is rounded and normalized.

For example:

\[
(4.734612 \times 10^3) \times (5.417242 \times 10^5) = (4.734612 \times 5.417242) \times 10^{3+5}
\]

\[
= 25.648538980104 \times 10^8 \quad \text{(true product)}
\]

\[
= 25.64854 \times 10^8 \quad \text{(after rounding)}
\]

\[
= 2.564854 \times 10^9 \quad \text{(after normalization)}
\]

Special operations:

- \( x \times (\pm 0) = \pm 0 \)
- \( x \times (\pm \infty) = \pm \infty \)
- \( \pm 0 \times (\pm \infty) = NaN \)
- \( x \times NaN = NaN \)
In this appendix the RTL code of all the modules is presented as well as the C code that was used.

For simplcity reasons, the following code snippet with the library imports has been removed from all the individual files.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

B.1 Floating Point Subadder

entity subadder is
  Port ( 
    clk : in std_logic;
    x : in STD_LOGIC_VECTOR (31 downto 0);
    y : in STD_LOGIC_VECTOR (31 downto 0);
    z : out STD_LOGIC_VECTOR (31 downto 0)
  );
end subadder;

architecture Behavioral of subadder is
  constant NaN : std_logic_vector(30 downto 0) := (others => '1');
  constant inf : std_logic_vector(30 downto 0) := "111" & X"f80_0000";
begin
  process(clk)
  variable x_mantissa : STD_LOGIC_VECTOR (24 downto 0);
  variable x_exponent : STD_LOGIC_VECTOR (7 downto 0);
  variable x_sign : STD_LOGIC;
  variable y_mantissa : STD_LOGIC_VECTOR (24 downto 0);
  variable y_exponent : STD_LOGIC_VECTOR (7 downto 0);
  variable y_sign : STD_LOGIC;
  variable mantissa_sum: std_logic_vector (24 downto 0);
  variable z_mantissa : STD_LOGIC_VECTOR (22 downto 0);
  variable z_exponent : STD_LOGIC_VECTOR (7 downto 0);
  variable z_sign : STD_LOGIC;
  variable temp_result : STD_LOGIC_VECTOR (31 downto 0):=(others =>'0');
  variable flag : std_logic;
  variable diff : unsigned (7 downto 0);
  begin
```
if rising_edge(clk) then
    x_mantissa := "01" & x(22 downto 0);
    x_exponent := x(30 downto 23);
    x_sign := x(31);
    y_mantissa := "01" & y(22 downto 0);
    y_exponent := y(30 downto 23);
    y_sign := y(31);
    flag := x_sign xor y_sign;
-- Special Cases
if (x_exponent=x"00" or y_exponent=x"00") then
    report "case_0";
    if x_exponent=x"00" then
        -- 0 +/- X = +/-X
        temp_result(30 downto 0) := y_exponent & x_mantissa(22 downto 0);
        temp_result(31) := x_sign;
    else
        -- X +/- 0 = X
        temp_result(31 downto 0) := x_sign & x_exponent & x_mantissa(22 downto 0);
    end if;
elsif (x_exponent=x"ff" or y_exponent=x"ff") then
    report "case_1";
    if (x_exponent & x_mantissa(22 downto 0))= NaN or (y_exponent & y_mantissa(22 downto 0))= NaN or flag='1' then
        -- inf = -inf = nan, nan +/- X = nan
        temp_result(30 downto 0) := NaN;
    else
        -- inf +/- X = inf
        temp_result(31 downto 0) := inf;
    end if;
    temp_result(31) := x_sign;
else
    if unsigned(x_exponent) > unsigned(y_exponent) then
        diff := unsigned(x_exponent) - unsigned(y_exponent);
        y_mantissa := std_logic_vector(shift_right(unsigned(y_mantissa),
            to_integer(diff)))
        z_exponent := x_exponent;
    else
        diff := unsigned(y_exponent) - unsigned(x_exponent);
        x_mantissa := std_logic_vector(shift_right(unsigned(x_mantissa),
            to_integer(diff)))
        z_exponent := y_exponent;
    end if;
    if flag = '0' then
        mantissa_sum := std_logic_vector(unsigned(x_mantissa) + unsigned(
            y_mantissa));
        z_sign := x_sign;
    elsif unsigned(x_mantissa) >= unsigned(y_mantissa) then
        mantissa_sum := std_logic_vector(unsigned(x_mantissa) - unsigned(
            y_mantissa));
        z_sign := x_sign;
    else
        mantissa_sum := std_logic_vector(unsigned(y_mantissa) - unsigned(
            x_mantissa));
        z_sign := y_sign;
    end if;
z_sign := y_sign;
end if;

if mantissa_sum = '0'&X"000000" then
  z_mantissa := (others => '0');
  z_exponent := (others => '0');
elsif mantissa_sum(24)='1' then
  z_mantissa := mantissa_sum(23 downto 1);
  z_exponent := std_logic_vector(unsigned(z_exponent)+1);
elsif mantissa_sum(23)='0' then
  for i in 22 downto 1 loop   --find position of the leading 1
    mantissa_sum := std_logic_vector(shift_left(unsigned(mantissa_sum) ,1)
  end loop;
  z_exponent := std_logic_vector(unsigned(z_exponent)- 1);
  z_mantissa := mantissa_sum(22 downto 0);
  if mantissa_sum(23) = '1' then
    exit;
  end if;
else
  z_mantissa := mantissa_sum(22 downto 0);
end if;

temp_result(22 downto 0) := z_mantissa;
temp_result(30 downto 23) := z_exponent;
temp_result(31) := z_sign;
end if;
z <= temp_result;
end if;
end process;
end Behavioral;
B.2 Floating Point Multiplier

entity multiplier is
  Port (  
    clk : in std_logic;
    a : in STD_LOGIC_VECTOR (31 downto 0);
    b : in STD_LOGIC_VECTOR (31 downto 0);
    result : out STD_LOGIC_VECTOR (31 downto 0) 
  );
end multiplier;

architecture Behavioral of multiplier is
begin
  process(clk)
  variable a_mantissa : STD_LOGIC_VECTOR (22 downto 0);
  variable a_exponent : STD_LOGIC_VECTOR (7 downto 0);
  variable a_sign : STD_LOGIC;
  variable b_mantissa : STD_LOGIC_VECTOR (22 downto 0);
  variable b_exponent : STD_LOGIC_VECTOR (7 downto 0);
  variable b_sign : STD_LOGIC;
  variable result_mantissa : STD_LOGIC_VECTOR (22 downto 0);
  variable result_exponent : STD_LOGIC_VECTOR (7 downto 0);
  variable result_sign : STD_LOGIC;
  variable aux : STD_LOGIC;
  variable aux2 : STD_LOGIC_VECTOR (47 downto 0);
  variable exponent_sum : STD_LOGIC_VECTOR (8 downto 0);
  variable temp_result : STD_LOGIC_VECTOR (31 downto 0);
  begin
    if rising_edge(clk) then
      a_mantissa := x(22 downto 0);
      a_exponent := x(30 downto 23);
      a_sign := x(31);
      b_mantissa := y(22 downto 0);
      b_exponent := y(30 downto 23);
      b_sign := y(31);
      temp_result(30 downto 0) := (others => '1');
    elsif a_mantissa /= 0 or b_mantissa /= 0 then
      temp_result(30 downto 0) := (others => '1');
    else
      aux2 := ('1' & a_mantissa) * ('1' & b_mantissa);
      aux := aux2(47);
      temp_result := X"f0000000";
    end if;
  end if;
end process(clk);
FLOATINGPOINTMULTIPLIER

>= 2, shift left and add one to exponent
result_mantissa := aux2(46 downto 24) + aux2(23); -- with rounding
else
result_mantissa := aux2(45 downto 23) + aux2(22); -- with rounding
end if;
-- calculate exponent
exponent_sum := ('0' & a_exponent) + ('0' & b_exponent) + aux - 127;

if (exponent_sum(8)='1') then
  if (exponent_sum(7)='0') then -- overflow
    result_exponent := "11111111";
    result_mantissa := (others => '0');
    result_sign := a_sign xor b_sign;
  else -- underflow
    result_exponent := (others => '0');
    result_mantissa := (others => '0');
    result_sign := '0';
  end if;
else -- Ok
  result_exponent := exponent_sum(7 downto 0);
  result_sign := a_sign xor b_sign;
end if;
temp_result(22 downto 0) := result_mantissa;
temp_result(30 downto 23) := z_exponent;
temp_result(31) := a_sign xor b_sign;
end if;
result <= temp_result;
end process;
end Behavioral;
B.3 Dot Product

**entity** dot_product **is**

Port ( clk : in STD_LOGIC;
    point1x : in STD_LOGIC_VECTOR (31 downto 0);
    point1y : in STD_LOGIC_VECTOR (31 downto 0);
    point1z : in STD_LOGIC_VECTOR (31 downto 0);
    point2x : in STD_LOGIC_VECTOR (31 downto 0);
    point2y : in STD_LOGIC_VECTOR (31 downto 0);
    point2z : in STD_LOGIC_VECTOR (31 downto 0);
    result : out STD_LOGIC_VECTOR (31 downto 0)
);  
end dot_product;

**architecture** Behavioral of dot_product **is**

    signal in1_c : STD_LOGIC_VECTOR(31 downto 0);
    signal in2_c : STD_LOGIC_VECTOR(31 downto 0);
    signal in1_c1 : STD_LOGIC_VECTOR(31 downto 0);
    signal in1_c2 : STD_LOGIC_VECTOR(31 downto 0);
    signal in2_c1 : STD_LOGIC_VECTOR(31 downto 0);
    signal in2_c2 : STD_LOGIC_VECTOR(31 downto 0);

    COMPONENT ip_mul
        PORT (  
            aclk : IN STD_LOGIC;
            s_axis_a_tvalid : IN STD_LOGIC;
            s_axis_a_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            s_axis_b_tvalid : IN STD_LOGIC;
            s_axis_b_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            m_axis_result_tvalid : OUT STD_LOGIC;
            m_axis_result_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
    END COMPONENT;

begin
    --architecture
    shift1_1 : entity work.shift_1
        PORT MAP (  
            D => point1y ,
            CLK => clk ,
            Q => in1_c1
        );
    
    shift3_1 : entity work.shift_3
        PORT MAP (  
            D => point1z ,
            CLK => clk ,
            Q => in2_c1
        );
    
    shift1_2 : entity work.shift_1
        PORT MAP (  
            D => point2y ,
            CLK => clk ,

    ...

64
B.3. DOT PRODUCT

Q => in1_c2
;

shift3_2 : entity work.shift_3
      PORT MAP (  
        D => point2z ,  
        CLK => clk ,  
        Q => in2_c2
      );
-- result = A * B
in0 : entity work.multiplier
      PORT MAP (  
        clk => clk ,  
        a => point1x ,  
        b => point2x ,  
        result => in1_c
      );
-- result = (A*B) + C
in1 : entity work.mul_add
      PORT MAP (  
        clk => clk ,  
        a => in1_c1 ,  
        b => in1_c2 ,  
        c => in1_c ,  
        result => in2_c
      );

in2 : entity work.mul_add
      PORT MAP (  
        clk => clk ,  
        a => in2_c1 ,  
        b => in2_c2 ,  
        c => in2_c ,  
        result => result
      );
end Behavioral;
B.4 Cross Product

B.4.1 Cross Element

```vhdl
entity cross_element is
  Port (
    clk : in STD_LOGIC;
    vec11 : in STD_LOGIC_VECTOR (31 downto 0);
    vec12 : in STD_LOGIC_VECTOR (31 downto 0);
    vec21 : in STD_LOGIC_VECTOR (31 downto 0);
    vec22 : in STD_LOGIC_VECTOR (31 downto 0);
    result : out STD_LOGIC_VECTOR (31 downto 0)
  );
end cross_element;

architecture Behavioral of cross_element is
  signal mul1_a : STD_LOGIC_VECTOR(31 downto 0);
  signal mul1_b : STD_LOGIC_VECTOR(31 downto 0);
  signal mul1_c : STD_LOGIC_VECTOR(31 downto 0);
begin
  shift1_0 : entity work.shift_1
  PORT MAP (  
    D => vec11,  
    CLK => clk,  
    Q => mul1_a
  );

  shift1_1 : entity work.shift_1
  PORT MAP (  
    D => vec21,  
    CLK => clk,  
    Q => mul1_b
  );

  -- result = A*B
  mul0 : entity work.multiplier
  PORT MAP (  
    clk => clk,  
    a => vec12,  
    b => vec22,  
    result => mul1_c
  );

  -- result = (A*B) - C
  mul1 : entity work.mul_sub
  PORT MAP (  
    clk => clk,  
    a => mul1_a,  
    b => mul1_b,  
    c => mul1_c,  
    result => result
  );
end Behavioral;
```
B.4.2 Cross Product

entity cross_product is
  Port (  
    clk : in STD_LOGIC;
    point1x : in STD_LOGIC_VECTOR (31 downto 0);
    point1y : in STD_LOGIC_VECTOR (31 downto 0);
    point1z : in STD_LOGIC_VECTOR (31 downto 0);
    point2x : in STD_LOGIC_VECTOR (31 downto 0);
    point2y : in STD_LOGIC_VECTOR (31 downto 0);
    point2z : in STD_LOGIC_VECTOR (31 downto 0);
    resultx : out STD_LOGIC_VECTOR (31 downto 0);
    resulty : out STD_LOGIC_VECTOR (31 downto 0);
    resultz : out STD_LOGIC_VECTOR (31 downto 0)
  );
end cross_product;

architecture Behavioral of cross_product is
begin
  elem0: entity work.cross_element
    port map (  
      clk => clk ,
      vec11 => point1y ,
      vec12 => point1z ,
      vec21 => point2z ,
      vec22 => point2y ,
      result => resultx
    );

  elem1: entity work.cross_element
    port map (  
      clk => clk ,
      vec11 => point1z ,
      vec12 => point1x ,
      vec21 => point2x ,
      vec22 => point2z ,
      result => resulty
    );

  elem2: entity work.cross_element
    port map (  
      clk => clk ,
      vec11 => point1x ,
      vec12 => point1y ,
      vec21 => point2y ,
      vec22 => point2x ,
      result => resultz
    );
end Behavioral;
B.5 Reciprocal Square Root

B.5.1 Newton - Raphson Element

entity newton_raphson_sqrt is
  Port (
    clk : in std_logic;
    init_estimate : in std_logic_vector(31 downto 0);
    input_number : in std_logic_vector(31 downto 0);
    result : out std_logic_vector(31 downto 0)
  );
end newton_raphson_sqrt;

architecture Behavioral of newton_raphson_sqrt is
  constant THREE : std_logic_vector(31 downto 0) := X"40400000";
  signal mul1_result: std_logic_vector(31 downto 0);
  signal mul0_result: std_logic_vector(31 downto 0);
  signal mul2_result: std_logic_vector(31 downto 0);
  signal subadd_result: std_logic_vector(31 downto 0);
  signal input_reg0: std_logic_vector(31 downto 0);
  signal input_reg1: std_logic_vector(31 downto 0);
  signal input_reg2: std_logic_vector(31 downto 0);
  signal input_reg3: std_logic_vector(31 downto 0);
  signal input_reg4: std_logic_vector(31 downto 0);
begin
  mul0 : entity work.multiplier
  port map(
    clk => clk,
    x => init_estimate,
    y => init_estimate,
    z => mul0_result
  );
  mul1 : entity work.multiplier
  port map(
    clk => clk,
    x => mul0_result,
    y => input_reg4,
    z => mul1_result
  );
  mul2 : entity work.multiplier
  port map(
    clk => clk,
    y => subadd_result,
    x => input_reg2,
    z => mul2_result
  );
  subadd : entity work.sub
  port map(
    clk => clk,
    x => THREE,
    y => mul1_result,
    z => subadd_result
  );
result(31) <= mul2_result(31);
result(30 downto 23) <= std_logic_vector(unsigned(mul2_result(30 downto 23)) - "00000001");
result(22 downto 0) <= mul2_result(22 downto 0);

process(clk)
begin
  if rising_edge(clk) then
    input_reg0 <= init_estimate;
    input_reg1 <= input_reg0;
    input_reg2 <= input_reg1;
    input_reg3 <= input_number;
    input_reg4 <= input_reg3;
  end if;
end process;
end Behavioral;
entity reciprocal_square_root is
  Port
  clk : in std_logic;
  input_number : in std_logic_vector(31 downto 0);
  result : out std_logic_vector(31 downto 0)
); end reciprocal_square_root;

architecture Behavioral of reciprocal_square_root is
  constant C0 : std_logic_vector (31 downto 0) := X"5f3759df";
  constant C0 : std_logic_vector (31 downto 0) := X"5f375a86";
  signal estimate : std_logic_vector(31 downto 0);
  signal input_reg0 : std_logic_vector(31 downto 0);
  signal input_reg1 : std_logic_vector(31 downto 0);
  signal input_reg2 : std_logic_vector(31 downto 0);
  signal input_reg3 : std_logic_vector(31 downto 0);
  signal input_reg4 : std_logic_vector(31 downto 0);
  signal NR_result0 : std_logic_vector(31 downto 0);
begin NR0 : entity work.newton_raphson_sqrt
  port map(
    clk => clk,
    init_estimate => estimate,
    input_number => input_number,
    result => NR_result0
  );
NR1 : entity work.newton_raphson_sqrt
  port map(
    clk => clk,
    init_estimate => NR_result0,
    input_number => input_reg3,
    result => result
  );

  estimate <= std_logic_vector(unsigned(C0) - unsigned('0' & input_number(31 downto 1))); process(clk)
begin
  if rising_edge(clk) then
    input_reg3 <= input_reg2;
    input_reg2 <= input_reg1;
    input_reg1 <= input_reg0;
    input_reg0 <= input_number;
  end if;
end process;
end Behavioral;
B.6 Euclidean Distance

entity reciprocal_distance is
    Port (  
        clk : in std_logic;
        vec1_x : in std_logic_vector(31 downto 0);
        vec1_y : in std_logic_vector(31 downto 0);
        vec1_z : in std_logic_vector(31 downto 0);
        vec2_x : in std_logic_vector(31 downto 0);
        vec2_y : in std_logic_vector(31 downto 0);
        vec2_z : in std_logic_vector(31 downto 0);
        result : out std_logic_vector(31 downto 0)
    );
end reciprocal_distance;

architecture Behavioral of reciprocal_distance is
-- signals --
signal sub0_in1 : std_logic_vector(31 downto 0) := (others => '0');
signal sub0_in2 : std_logic_vector(31 downto 0) := (others => '0');
signal sub0_result : std_logic_vector(31 downto 0) := (others => '0');
signal sub1_in1 : std_logic_vector(31 downto 0) := (others => '0');
signal sub1_in2 : std_logic_vector(31 downto 0) := (others => '0');
signal sub1_result : std_logic_vector(31 downto 0) := (others => '0');
signal sub2_in1 : std_logic_vector(31 downto 0) := (others => '0');
signal sub2_in2 : std_logic_vector(31 downto 0) := (others => '0');
signal sub2_result : std_logic_vector(31 downto 0) := (others => '0');
-- Dot product signals --
signal dot_input_x : std_logic_vector (31 downto 0) := (others => '0');
signal dot_input_y : std_logic_vector (31 downto 0) := (others => '0');
signal dot_input_z : std_logic_vector (31 downto 0) := (others => '0');
signal dot_result : std_logic_vector (31 downto 0) := (others => '0');
signal dot_result2 : std_logic_vector (31 downto 0) := (others => '0');
-- RSR signals --
signal rsr_input : std_logic_vector(31 downto 0) := (others => '0');
signal rsr_result : std_logic_vector(31 downto 0) := (others => '0');
begin
-- architecture

sub0 : entity work.sub
port map(
          clk => clk ,
          x => sub0_in1 ,
          y => sub0_in2 ,
          z => sub0_result 
      );

sub1 : entity work.sub
port map(
          clk => clk ,
          x => sub1_in1 ,
          y => sub1_in2 ,
          z => sub1_result 
      );

sub2 : entity work.sub
port map(
clk => clk,
x => sub2_in1,
y => sub2_in2,
z => sub2_result);

dot : entity work.dot_product
PORT MAP (  
  clk => clk,
  point1x => dot_input_x,
  point1y => dot_input_y,
  point1z => dot_input_z,
  point2x => dot_input_x,
  point2y => dot_input_y,
  point2z => dot_input_z,
  result => rsr_input);

rsr : entity work.reciprocal_square_root
PORT MAP (  
  clk => clk,
  input_number => rsr_input,
  result => result);

dot_input_x <= sub0_result;
dot_input_y <= sub1_result;
dot_input_z <= sub2_result;

sub0_in1 <= vec1_x;
sub0_in2 <= vec2_x;
sub1_in1 <= vec1_y;
sub1_in2 <= vec2_y;
sub2_in1 <= vec1_z;
sub2_in2 <= vec2_z;
end Behavioral;
B.7 Normal Vector

entity reciprocal_normal is
  Port (  
    clk : in std_logic;  
    vec_x : in std_logic_vector(31 downto 0);  
    vec_y : in std_logic_vector(31 downto 0);  
    vec_z : in std_logic_vector(31 downto 0);  
    result : out std_logic_vector(31 downto 0)  
  );
end reciprocal_normal;

architecture Behavioral of reciprocal_normal is
  -- Dot product signals --
  signal dot_input_x : std_logic_vector (31 downto 0) := (others => '0');
  signal dot_input_y : std_logic_vector (31 downto 0) := (others => '0');
  signal dot_input_z : std_logic_vector (31 downto 0) := (others => '0');
  signal dot_result : std_logic_vector (31 downto 0) := (others => '0');
  signal dot_result2 : std_logic_vector (31 downto 0) := (others => '0');

  -- RSR signals --
  signal rsr_input : std_logic_vector(31 downto 0) := (others => '0');
  signal rsr_result : std_logic_vector(31 downto 0) := (others => '0');
begin
  --architecture
  dot : entity work.dot_product
    PORT MAP (  
      clk => clk,  
      point1x => dot_input_x,  
      point1y => dot_input_y,  
      point1z => dot_input_z,  
      point2x => dot_input_x,  
      point2y => dot_input_y,  
      point2z => dot_input_z,  
      result => rsr_input  
    );

  rsr : entity work.reciprocal_square_root
    port map (  
      clk => rsr_clk,  
      input_number => rsr_input,  
      result => rsr_result  
    );

  dot_input_x <= vec_x;
  dot_input_y <= vec_y;
  dot_input_z <= vec_z;
end Behavioral;
B.8 Atan

B.8.1 Single Input Atan

```vhdl
-- atan = [(A*x^2 + B)*x + C]*x

entity Atan is
  Port (  
    x : in std_logic_vector(31 downto 0); 
    clk : in std_logic; 
    result : out std_logic_vector(31 downto 0) 
  );
end Atan;

architecture Behavioral of Atan is

  -- Multiplier signals
  signal mul0_in1 : std_logic_vector (31 downto 0) := (others => '0');
  signal mul0_in2 : std_logic_vector (31 downto 0) := (others => '0');
  signal mul0_result : std_logic_vector (31 downto 0) := (others => '0');
  signal mul1_in1 : std_logic_vector (31 downto 0) := (others => '0');
  signal mul1_in2 : std_logic_vector (31 downto 0) := (others => '0');
  signal mul1_result : std_logic_vector (31 downto 0) := (others => '0');
  signal mul2_in1 : std_logic_vector (31 downto 0) := (others => '0');
  signal mul2_in2 : std_logic_vector (31 downto 0) := (others => '0');
  signal mul2_result : std_logic_vector (31 downto 0) := (others => '0');
  signal mul3_in1 : std_logic_vector (31 downto 0) := (others => '0');
  signal mul3_in2 : std_logic_vector (31 downto 0) := (others => '0');
  signal mul3_result : std_logic_vector (31 downto 0) := (others => '0');

  -- Sub/adder signals
  signal subadd0_in1 : std_logic_vector (31 downto 0) := (others => '0');
  signal subadd0_in2 : std_logic_vector (31 downto 0) := (others => '0');
  signal subadd0_result : std_logic_vector (31 downto 0) := (others => '0');
  signal subadd1_in1 : std_logic_vector (31 downto 0) := (others => '0');
  signal subadd1_in2 : std_logic_vector (31 downto 0) := (others => '0');
  signal subadd1_result : std_logic_vector (31 downto 0) := (others => '0');

  -- Registers
  signal x_reg4 : std_logic_vector (31 downto 0) := (others => '0');
  signal xx_reg1 : std_logic_vector (31 downto 0) := (others => '0');

  constant A : std_logic_vector (31 downto 0) := "3d9f_0777";
  constant B : std_logic_vector (31 downto 0) := "be93_2a9d";
  constant C : std_logic_vector (31 downto 0) := "3f7e_c43a";

COMPONENT shift_5
PORT (  
  D : IN STD_LOGIC_VECTOR(31 DOWNTO 0); 
  CLK : IN STD_LOGIC; 
  Q : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) 
);
END COMPONENT;
COMPONENT shift_2
PORT (  
  D : IN STD_LOGIC_VECTOR(31 DOWNTO 0) 
);
CLK : IN STD_LOGIC;
Q : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END COMPONENT;
begin --architecture

mul0 : entity work.multiplier
port map(
    clk => clk,
    x => mul0_in1,
    y => mul0_in2,
    z => mul0_result
);

mul1 : entity work.multiplier
port map(
    clk => clk,
    x => mul1_in1,
    y => mul1_in2,
    z => mul1_result
);

mul2 : entity work.multiplier
port map(
    clk => clk,
    x => mul2_in1,
    y => mul2_in2,
    z => mul2_result
);

mul3 : entity work.multiplier
port map(
    clk => clk,
    x => mul3_in1,
    y => mul3_in2,
    z => mul3_result
);

subadd0: entity work.adder
port map(
    clk => clk,
    x => subadd0_in1,
    y => subadd0_in2,
    z => subadd0_result
);

subadd1: entity work.adder
port map(
    clk => clk,
    x => subadd1_in1,
    y => subadd1_in2,
    z => subadd1_result
);

mul0_in1 <= x;
mul0_in2 <= x;
mul1_in1 <= mul0_result;
mul1_in2 <= A;
mul2_in1 <= subadd0_result;
mul2_in2 <= xx_reg1;

mul3_in1 <= subadd1_result;
mul3_in2 <= x_reg4;

subadd0_in1 <= mul1_result;
subadd0_in2 <= B;

subadd1_in1 <= mul2_result;
subadd1_in2 <= C;

result <= mul3_result;

shift5_0 : shift_5
PORT MAP (  
D => x,  
CLK => CLK,  
Q => x_reg4
);

shift2_0 : shift_2
PORT MAP (  
D => mul0_result,  
CLK => CLK,  
Q => xx_reg1
);
end Behavioral;
B.8.2 Two Input Atan

```
entity Atan2 is
  Port (
    x : in std_logic_vector(31 downto 0);
    y : in std_logic_vector(31 downto 0);
    clk : in std_logic;
    result : out std_logic_vector(31 downto 0)
  );
end Atan2;
```

```
architecture Behavioral of Atan2 is
  -- Simple Atan signals
  signal atan_result : std_logic_vector (31 downto 0) := (others => '0');
  signal atan_input : std_logic_vector (31 downto 0) := (others => '0');

  -- Sub/adder signals
  signal subadd0_in1 : std_logic_vector (31 downto 0) := (others => '0');
  signal subadd0_in2 : std_logic_vector (31 downto 0) := (others => '0');
  signal subadd0_result : std_logic_vector (31 downto 0) := (others => '0');
  signal subadd1_in1 : std_logic_vector (31 downto 0) := (others => '0');
  signal subadd1_in2 : std_logic_vector (31 downto 0) := (others => '0');
  signal subadd1_result : std_logic_vector (31 downto 0) := (others => '0');

  -- Divider signals
  signal div_input_1: std_logic_vector (31 downto 0) := (others => '0');
  signal div_input_2: std_logic_vector (31 downto 0) := (others => '0');
  signal div_result : std_logic_vector (31 downto 0) := (others => '0');

  -- Absolute value signals
  signal offset : std_logic_vector (31 downto 0) := (others => '0');
  signal abs_x: std_logic_vector (31 downto 0);
  signal abs_y: std_logic_vector (31 downto 0);
  signal flag_neg : std_logic := '0';
  signal flag_reversed : std_logic := '0';

  -- Registers
  signal x_sign : std_logic_vector(19 downto 0) := (others => '0');
  signal y_sign : std_logic_vector(19 downto 0) := (others => '0');
  signal flags : std_logic_vector (35 downto 0) := (others => '0');

  constant PI : std_logic_vector (31 downto 0) := X"4049_0fdb"; -- 3.14159265
  constant PI_2 : std_logic_vector (31 downto 0) := X"3fc9_0fdb"; -- 1.57079637
  constant PI_n : std_logic_vector (31 downto 0) := X"c049_0fdb"; -- -3.14159265
  constant PI_2n : std_logic_vector (31 downto 0) := X"bfc9_0fdb"; -- -1.57079637
  constant ZERO : std_logic_vector (31 downto 0) := X"0000_0000";
```

```
COMPONENT fp_divider
PORT ( 
  aclk : IN STD_LOGIC;
  s_axis_a_tvalid : IN STD_LOGIC;
  s_axis_a_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
  s_axis_b_tvalid : IN STD_LOGIC;
  s_axis_b_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
)
m_axis_result_tvalid : OUT STD_LOGIC;
m_axis_result_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
END COMPONENT;
begin
  --architecture
  atan : entity work.atan
  port map(
    x => atan_input,
    result => atan_result,
    clk => clk
  );

  subadd0: entity work.adder
  port map(
    clk => clk,
    x => subadd0_in1,
    y => subadd0_in2,
    z => subadd0_result
  );

  subadd1: entity work.adder
  port map(
    clk => clk,
    x => subadd1_in1,
    y => subadd1_in2,
    z => subadd1_result
  );

  div : fp_divider
  PORT MAP (
    aclk => clk,
    s_axis_a_tvalid => '1',
    s_axis_a_tdata => div_input_1,
    s_axis_b_tvalid => '1',
    s_axis_b_tdata => div_input_2,
    m_axis_result_tdata => div_result
  );

  offset <= PI when y_sign(19) = '0' else PI_n;

  flag_neg <= x(31) xor y(31);
  flag_reversed <= '0' when unsigned(abs_x) > unsigned(abs_y) else '1';

  x_sign(0) <= x(31);
  y_sign(0) <= y(31);

  abs_x <= '0' & x(30 downto 0);
  abs_y <= '0' & y(30 downto 0);

  div_input_1 <= abs_x when flag_reversed = '1' else abs_y;
  div_input_2 <= abs_y when flag_reversed = '1' else abs_x;

  atan_input <= div_result;
subadd1_in1 <= subadd0_result;
subadd1_in2 <= offset when x_sign(18) = '1' else ZERO;

result <= subadd1_result;

process (clk)
begin
  if rising_edge(clk) then
    flags <= std_logic_vector(shift_left(unsigned(flags),2));
    flags(1 downto 0) <= flag_reversed & flag_neg;
    x_sign(19 downto 1) <= std_logic_vector(shift_left(unsigned(x_sign(19 downto 1)),1));
    y_sign(19 downto 1) <= std_logic_vector(shift_left(unsigned(y_sign(19 downto 1)),1));
    case flags(35 downto 34) is
    when "00" =>
      subadd0_in1 <= ZERO;
      subadd0_in2 <= atan_result;
    when "01" =>
      subadd0_in1 <= ZERO;
      subadd0_in2 <= not atan_result(31) & atan_result(30 downto 0);
    when "10" =>
      subadd0_in1 <= PI_2;
      subadd0_in2 <= not atan_result(31) & atan_result(30 downto 0);
    when others =>
      subadd0_in1 <= atan_result;
      subadd0_in2 <= PI_2n;
    end case;
  end if;
end process;
end Behavioral;
B.9 Pair Features

```vhdl
entity pair_features is
  port ( clk : in std_logic;
         point1x : in std_logic_vector (31 downto 0);
         point1y : in std_logic_vector (31 downto 0);
         point1z : in std_logic_vector (31 downto 0);
         point2x : in std_logic_vector (31 downto 0);
         point2y : in std_logic_vector (31 downto 0);
         point2z : in std_logic_vector (31 downto 0);
         normal1x : in std_logic_vector (31 downto 0);
         normal1y : in std_logic_vector (31 downto 0);
         normal1z : in std_logic_vector (31 downto 0);
         normal2x : in std_logic_vector (31 downto 0);
         normal2y : in std_logic_vector (31 downto 0);
         normal2z : in std_logic_vector (31 downto 0);
         feature0 : out std_logic_vector (31 downto 0);
         feature1 : out std_logic_vector (31 downto 0);
         feature2 : out std_logic_vector (31 downto 0);
         feature3 : out std_logic_vector (31 downto 0) );
end pair_features;

architecture behavioral of pair_features is

-- signals for normal
signal normal0_x: std_logic_vector (31 downto 0) := (others => '0');
signal normal0_y: std_logic_vector (31 downto 0) := (others => '0');
signal normal0_z: std_logic_vector (31 downto 0) := (others => '0');
signal normal0_result: std_logic_vector (31 downto 0) := (others => '0');

-- signals for distance
signal dist0_1x: std_logic_vector (31 downto 0) := (others => '0');
signal dist0_1y: std_logic_vector (31 downto 0) := (others => '0');
signal dist0_1z: std_logic_vector (31 downto 0) := (others => '0');
signal dist0_2x: std_logic_vector (31 downto 0) := (others => '0');
signal dist0_2y: std_logic_vector (31 downto 0) := (others => '0');
signal dist0_2z: std_logic_vector (31 downto 0) := (others => '0');
signal dist0_result: std_logic_vector (31 downto 0) := (others => '0');

-- signals for atan2
signal atan0_x: std_logic_vector (31 downto 0) := (others => '0');
signal atan0_y: std_logic_vector (31 downto 0) := (others => '0');
signal atan0_result: std_logic_vector (31 downto 0) := (others => '0');

-- signals for cross 0
signal cross0_1x: std_logic_vector (31 downto 0) := (others => '0');
signal cross0_1y: std_logic_vector (31 downto 0) := (others => '0');
signal cross0_1z: std_logic_vector (31 downto 0) := (others => '0');
signal cross0_2x: std_logic_vector (31 downto 0) := (others => '0');
signal cross0_2y: std_logic_vector (31 downto 0) := (others => '0');
signal cross0_2z: std_logic_vector (31 downto 0) := (others => '0');
signal cross0_resultx: std_logic_vector (31 downto 0) := (others => '0');
signal cross0_resulty: std_logic_vector (31 downto 0) := (others => '0');
signal cross0_resultz: std_logic_vector (31 downto 0) := (others => '0');

-- signals for cross 1
signal cross1_1x: std_logic_vector (31 downto 0) := (others => '0');
signal cross1_1y: std_logic_vector (31 downto 0) := (others => '0');
signal cross1_1z: std_logic_vector (31 downto 0) := (others => '0');

end behavioral;
```
signal cross1_2x: std_logic_vector (31 downto 0) := (others => '0');
signal cross1_2y: std_logic_vector (31 downto 0) := (others => '0');
signal cross1_2z: std_logic_vector (31 downto 0) := (others => '0');
signal cross1_resultx: std_logic_vector (31 downto 0) := (others => '0');
signal cross1_resulty: std_logic_vector (31 downto 0) := (others => '0');
signal cross1_resultz: std_logic_vector (31 downto 0) := (others => '0');

-- signal for dot product 0
signal dot0_1x: std_logic_vector (31 downto 0) := (others => '0');
signal dot0_1y: std_logic_vector (31 downto 0) := (others => '0');
signal dot0_1z: std_logic_vector (31 downto 0) := (others => '0');
signal dot0_2x: std_logic_vector (31 downto 0) := (others => '0');
signal dot0_2y: std_logic_vector (31 downto 0) := (others => '0');
signal dot0_2z: std_logic_vector (31 downto 0) := (others => '0');
signal dot0_result: std_logic_vector (31 downto 0) := (others => '0');

-- signal for dot product 1
signal dot1_1x: std_logic_vector (31 downto 0) := (others => '0');
signal dot1_1y: std_logic_vector (31 downto 0) := (others => '0');
signal dot1_1z: std_logic_vector (31 downto 0) := (others => '0');
signal dot1_2x: std_logic_vector (31 downto 0) := (others => '0');
signal dot1_2y: std_logic_vector (31 downto 0) := (others => '0');
signal dot1_2z: std_logic_vector (31 downto 0) := (others => '0');
signal dot1_result: std_logic_vector (31 downto 0) := (others => '0');

-- signal for dot product 2
signal dot2_1x: std_logic_vector (31 downto 0) := (others => '0');
signal dot2_1y: std_logic_vector (31 downto 0) := (others => '0');
signal dot2_1z: std_logic_vector (31 downto 0) := (others => '0');
signal dot2_2x: std_logic_vector (31 downto 0) := (others => '0');
signal dot2_2y: std_logic_vector (31 downto 0) := (others => '0');
signal dot2_2z: std_logic_vector (31 downto 0) := (others => '0');
signal dot2_result: std_logic_vector (31 downto 0) := (others => '0');

-- signal for dot product 3
signal dot3_1x: std_logic_vector (31 downto 0) := (others => '0');
signal dot3_1y: std_logic_vector (31 downto 0) := (others => '0');
signal dot3_1z: std_logic_vector (31 downto 0) := (others => '0');
signal dot3_2x: std_logic_vector (31 downto 0) := (others => '0');
signal dot3_2y: std_logic_vector (31 downto 0) := (others => '0');
signal dot3_2z: std_logic_vector (31 downto 0) := (others => '0');
signal dot3_result: std_logic_vector (31 downto 0) := (others => '0');

-- signal for dot product 4
signal dot4_1x: std_logic_vector (31 downto 0) := (others => '0');
signal dot4_1y: std_logic_vector (31 downto 0) := (others => '0');
signal dot4_1z: std_logic_vector (31 downto 0) := (others => '0');
signal dot4_2x: std_logic_vector (31 downto 0) := (others => '0');
signal dot4_2y: std_logic_vector (31 downto 0) := (others => '0');
signal dot4_2z: std_logic_vector (31 downto 0) := (others => '0');
signal dot4_result: std_logic_vector (31 downto 0) := (others => '0');

-- signals for multiplier 0
signal mul0_x: std_logic_vector (31 downto 0) := (others => '0');
signal mul0_y: std_logic_vector (31 downto 0) := (others => '0');
signal mul0_z: std_logic_vector (31 downto 0) := (others => '0');

-- signals for multiplier 1
signal mul1_x: std_logic_vector (31 downto 0) := (others => '0');
signal mul1_y: std_logic_vector (31 downto 0) := (others => '0');
signal mul1_z: std_logic_vector (31 downto 0) := (others => '0');

-- signals for multiplier 2
APPENDIX B. CODES

```vhdl
signal mul2_x: std_logic_vector (31 downto 0) := (others => '0');
signal mul2_y: std_logic_vector (31 downto 0) := (others => '0');
signal mul2_z: std_logic_vector (31 downto 0) := (others => '0');
-- signals for multiplier 3
signal mul3_x: std_logic_vector (31 downto 0) := (others => '0');
signal mul3_y: std_logic_vector (31 downto 0) := (others => '0');
signal mul3_z: std_logic_vector (31 downto 0) := (others => '0');
-- signals for multiplier 4
signal mul4_x: std_logic_vector (31 downto 0) := (others => '0');
signal mul4_y: std_logic_vector (31 downto 0) := (others => '0');
signal mul4_z: std_logic_vector (31 downto 0) := (others => '0');
-- signals for subtractor 0
signal sub0_x: std_logic_vector (31 downto 0) := (others => '0');
signal sub0_y: std_logic_vector (31 downto 0) := (others => '0');
signal sub0_z: std_logic_vector (31 downto 0) := (others => '0');
-- signals for subtractor 1
signal sub1_x: std_logic_vector (31 downto 0) := (others => '0');
signal sub1_y: std_logic_vector (31 downto 0) := (others => '0');
signal sub1_z: std_logic_vector (31 downto 0) := (others => '0');
-- signals for subtractor 2
signal sub2_x: std_logic_vector (31 downto 0) := (others => '0');
signal sub2_y: std_logic_vector (31 downto 0) := (others => '0');
signal sub2_z: std_logic_vector (31 downto 0) := (others => '0');
-- signals for temp normals
signal n1x: std_logic_vector (31 downto 0) := (others => '0');
signal n1y: std_logic_vector (31 downto 0) := (others => '0');
signal n1z: std_logic_vector (31 downto 0) := (others => '0');
signal n2x: std_logic_vector (31 downto 0) := (others => '0');
signal n2y: std_logic_vector (31 downto 0) := (others => '0');
signal n2z: std_logic_vector (31 downto 0) := (others => '0');
signal n01x: std_logic_vector (31 downto 0) := (others => '0');
signal n01y: std_logic_vector (31 downto 0) := (others => '0');
signal n01z: std_logic_vector (31 downto 0) := (others => '0');
signal n02x: std_logic_vector (31 downto 0) := (others => '0');
signal n02y: std_logic_vector (31 downto 0) := (others => '0');
signal n02z: std_logic_vector (31 downto 0) := (others => '0');
signal n11x: std_logic_vector (31 downto 0) := (others => '0');
signal n11y: std_logic_vector (31 downto 0) := (others => '0');
signal n11z: std_logic_vector (31 downto 0) := (others => '0');
signal n12x: std_logic_vector (31 downto 0) := (others => '0');
signal n12y: std_logic_vector (31 downto 0) := (others => '0');
signal n12z: std_logic_vector (31 downto 0) := (others => '0');
signal n21x: std_logic_vector (31 downto 0) := (others => '0');
signal n21y: std_logic_vector (31 downto 0) := (others => '0');
signal n21z: std_logic_vector (31 downto 0) := (others => '0');
signal n22x: std_logic_vector (31 downto 0) := (others => '0');
signal n22y: std_logic_vector (31 downto 0) := (others => '0');
signal n22z: std_logic_vector (31 downto 0) := (others => '0');
signal n32x: std_logic_vector (31 downto 0) := (others => '0');
signal n32y: std_logic_vector (31 downto 0) := (others => '0');
signal n32z: std_logic_vector (31 downto 0) := (others => '0');
-- signals for theta
signal theta1: std_logic_vector (31 downto 0) := (others => '0');
signal theta2: std_logic_vector (31 downto 0) := (others => '0');
-- signals for v vector
```

82
B.9. PAIR FEATURES

```vhdl
signal vx: std_logic_vector (31 downto 0) := (others => '0');
signal vy: std_logic_vector (31 downto 0) := (others => '0');
signal vz: std_logic_vector (31 downto 0) := (others => '0');

-- signals for w vector
signal wx: std_logic_vector (31 downto 0) := (others => '0');
signal wy: std_logic_vector (31 downto 0) := (others => '0');
signal wz: std_logic_vector (31 downto 0) := (others => '0');

-- signals for d2p1
signal d2p1_x: std_logic_vector (31 downto 0) := (others => '0');
signal d2p1_y: std_logic_vector (31 downto 0) := (others => '0');
signal d2p1_z: std_logic_vector (31 downto 0) := (others => '0');
signal d2p1_x: std_logic_vector (31 downto 0) := (others => '0');

-- intermediate signals
signal f0: std_logic_vector (31 downto 0) := (others => '0');
signal f1: std_logic_vector (31 downto 0) := (others => '0');
signal f2: std_logic_vector (31 downto 0) := (others => '0');
signal f3: std_logic_vector (31 downto 0) := (others => '0');

begin
  -- architecture reciprocal normal
  normal0 : entity work.fp_normal
  port map (clk => clk,
            vec_x => normal0_x,
            vec_y => normal0_y,
            vec_z => normal0_z,
            result => normal0_result);

  -- architecture reciprocal distance
  dist0 : entity work.fp_dist
  port map (clk => clk,
            vec1_x => dist0_1x,
            vec1_y => dist0_1y,
            vec1_z => dist0_1z,
            vec2_x => dist0_2x,
            vec2_y => dist0_2y,
            vec2_z => dist0_2z);
```
result => dist0_result
);

-- atan2
atan0 : entity work.fp_atan
port map (
x => atan0_x,
y => atan0_y,
clk => clk,
result => atan0_result
);

-- cross product
cross_0 : entity work.fp_cross
port map (
clk => clk,
point1x => cross0_1x,
point1y => cross0_1y,
point1z => cross0_1z,
point2x => cross0_2x,
point2y => cross0_2y,
point2z => cross0_2z,
resultx => cross0_resultx,
resulty => cross0_resulty,
resultz => cross0_resultz
);
cross_1 : entity work.fp_cross
port map (
clk => clk,
point1x => cross1_1x,
point1y => cross1_1y,
point1z => cross1_1z,
point2x => cross1_2x,
point2y => cross1_2y,
point2z => cross1_2z,
resultx => cross1_resultx,
resulty => cross1_resulty,
resultz => cross1_resultz
);

-- dot product
dot0 : entity work.fp_dot
port map (
clk => clk,
point1x => dot0_1x,
point1y => dot0_1y,
point1z => dot0_1z,
point2x => dot0_2x,
point2y => dot0_2y,
point2z => dot0_2z,
result => dot0_result
);
dot1 : entity work.fp_dot
port map (
clk => clk,
point1x => dot1_1x,
point1y => dot1_1y,
point1z => dot1_1z,
point2x => dot1_2x,
point2y => dot1_2y,
point2z => dot1_2z,
result => dot1_result);
dot2 : entity work.fp_dot
port map (
  clk => clk,
  point1x => dot2_1x,
  point1y => dot2_1y,
  point1z => dot2_1z,
  point2x => dot2_2x,
  point2y => dot2_2y,
  point2z => dot2_2z,
  result => dot2_result);
dot3 : entity work.fp_dot
port map (
  clk => clk,
  point1x => dot3_1x,
  point1y => dot3_1y,
  point1z => dot3_1z,
  point2x => dot3_2x,
  point2y => dot3_2y,
  point2z => dot3_2z,
  result => dot3_result);
dot4 : entity work.fp_dot
port map (
  clk => clk,
  point1x => dot4_1x,
  point1y => dot4_1y,
  point1z => dot4_1z,
  point2x => dot4_2x,
  point2y => dot4_2y,
  point2z => dot4_2z,
  result => dot4_result);
-- multipliers
mul0 : entity work.mul
port map (
  clk => clk,
  x => mul0_x,
  y => mul0_y,
  z => mul0_z);
mul1 : entity work.mul
port map (
  clk => clk,
  x => mul1_x,
  y => mul1_y,
  z => mul1_z);
mul2 : entity work.mul
port map (}
APPENDIX B. CODES

clk => clk,
x => mul2_x,
y => mul2_y,
z => mul2_z);

mul3 : entity work.mul
port map (  
clk => clk,
x => mul3_x,
y => mul3_y,
z => mul3_z  );

mul4 : entity work.mul
port map (  
clk => clk,
x => mul4_x,
y => mul4_y,
z => mul4_z  );

-- subtractors
sub0 : entity work.fp_sub
port map (  
clk => clk,
x => sub0_x,
y => sub0_y,
z => sub0_z  );

sub1 : entity work.fp_sub
port map (  
clk => clk,
x => sub1_x,
y => sub1_y,
z => sub1_z  );

sub2 : entity work.fp_sub
port map (  
clk => clk,
x => sub2_x,
y => sub2_y,
z => sub2_z  );

-- delays

delay_theta1_x : entity work.delay_theta
port map (  
d => normal1x ,
clk => clk ,
q => n1x  );

delay_theta1_y : entity work.delay_theta
port map (  
d => normal1y ,
clk => clk ,
q => n1y  );
delay_theta1_z : entity work.delay_theta
B.9. PAIR FEATURES

```vhdl
port map (  
d => normal1z,  
clk => clk,  
q => n1z  
);
delay_theta2_x : entity work.delay_theta
port map (  
d => normal2x,  
clk => clk,  
q => n2x  
);
delay_theta2_y : entity work.delay_theta
port map (  
d => normal2y,  
clk => clk,  
q => n2y  
);
delay_theta2_z : entity work.delay_theta
port map (  
d => normal2z,  
clk => clk,  
q => n2z  
);
delay_d2p1x : entity work.delay_d2p1
port map (  
d => sub0_z,  
clk => clk,  
q => d2p1_x  
);
delay_d2p1y : entity work.delay_d2p1
port map (  
d => sub1_z,  
clk => clk,  
q => d2p1_y  
);
delay_d2p1z : entity work.delay_d2p1
port map (  
d => sub2_z,  
clk => clk,  
q => d2p1_z  
);
delay_dd2p1x : entity work.delay_dd2p1
port map (  
d => d2p1_x,  
clk => clk,  
q => dd2p1_x  
);
delay_dd2p1y : entity work.delay_dd2p1
port map (  
d => d2p1_y,  
clk => clk,  
q => dd2p1_y  
);
delay_dd2p1z : entity work.delay_dd2p1
port map (  
```
APPENDIX B. CODES

d => d2p1_z,
clk => clk,
q => dd2p1_x
);
delay_rsr0_flag : entity work.delay_rsr0
port map (d => rsr_is_0,
clk => clk,
q => delay_rsr_is_0);
delay_v0_flag : entity work.delay_v0
port map (d => v_is_0,
clk => clk,
q => delay_v_is_0);
delay_feature1 : entity work.delay_f1
port map (d => f1,
clk => clk,
q => df1);
delay_feature2 : entity work.delay_f2
port map (d => f2,
clk => clk,
q => df2);
delay_feature3 : entity work.delay_f3
port map (d => dist0_result,
clk => clk,
q => feature3);
delay_vx : entity work.delay_v
port map (d => cross0_resultx,
clk => clk,
q => vx);
delay_vy : entity work.delay_v
port map (d => cross0_resulty,
clk => clk,
q => vy);
delay_vz : entity work.delay_v
port map (d => cross0_resultz,
clk => clk,
q => vz);
delay_atan_iny : entity work.delay_atan_in
port map (d => dot4_result,
clk => clk,
q => delayed_atan0_y
);
delay_n01x : entity work.delay_n0
port map (
  d => n1x,
  clk => clk,
  q => n01x
);
delay_n01y : entity work.delay_n0
port map (
  d => n1y,
  clk => clk,
  q => n01y
);
delay_n01z : entity work.delay_n0
port map (
  d => n1z,
  clk => clk,
  q => n01z
);
delay_n02x : entity work.delay_n0
port map (
  d => n2x,
  clk => clk,
  q => n02x
);
delay_n02y : entity work.delay_n0
port map (
  d => n2y,
  clk => clk,
  q => n02y
);
delay_n02z : entity work.delay_n0
port map (
  d => n2z,
  clk => clk,
  q => n02z
);
delay_n11x : entity work.delay_n1
port map (
  d => n11x,
  clk => clk,
  q => n12x
);
delay_n11y : entity work.delay_n1
port map (
  d => n11y,
  clk => clk,
  q => n12y
);
delay_n11z : entity work.delay_n1
port map (
  d => n11z,
  clk => clk,
q => n12z
);
delay_n2x : entity work.delay_n2
port map (  
d => n21x,  
clk => clk,  
q => n22x
);
delay_n2y : entity work.delay_n2
port map (  
d => n21y,  
clk => clk,  
q => n22y
);
delay_n2z : entity work.delay_n2
port map (  
d => n21z,  
clk => clk,  
q => n22z
);
delay_n21x : entity work.delay_n21
port map (  
d => n22x,  
clk => clk,  
q => n32x
);
delay_n21y : entity work.delay_n21
port map (  
d => n22y,  
clk => clk,  
q => n32y
);
delay_n21z : entity work.delay_n21
port map (  
d => n22z,  
clk => clk,  
q => n32z
);
-- signal assignment/connections  
sub0_x <= point2x;  
sub0_y <= point1x;

sub1_x <= point2y;  
sub1_y <= point1y;

sub2_x <= point2z;  
sub2_y <= point1z;

dist0_1x <= point1x;  
dist0_1y <= point1y;  
dist0_1z <= point1z;  
dist0_2x <= point2x;  
dist0_2y <= point2y;  
dist0_2z <= point2z;
dot0_1x <= n1x;
dot0_1y <= n1y;
dot0_1z <= n1z;
dot0_2x <= d2x;
dot0_2y <= d2y;
dot0_2z <= d2z;
mul0_x <= dot0_result;

-- theta1 = dot(n1, dp2p1)/f[3]
dot1_1x <= n2x;
dot1_1y <= n2y;
dot1_1z <= n2z;
mull_x <= dot1_result;
mul0_y <= dist0_result;
theta1 <= mul0_z;

-- theta2 = dot(n2, dp2p1)/f[3]
dot1_2x <= d2x;
dot1_2y <= d2y;
dot1_2z <= d2z;
mull_y <= dist0_result;
theta2 <= mul1_z;

is_zero <= delay_rsr_is_0(0) or delay_v_is_0(0);
rsr_is_0 <= "1" when unsigned(dist0_result(30 downto 0)) = unsigned(inf) else "0";
v_is_0 <= "1" when unsigned(normal0_result(30 downto 0)) = unsigned(inf) else "0";

-- if acos(|theta1|) > acos(|theta2|) -> swap n1 with n2 and f2 = theta2 else theta1
-- if |theta2| > |theta1| -> acos(|theta1|) > acos(|theta2|)
to_reverse <= '1' when unsigned(theta2(30 downto 0)) >= unsigned(theta1(30 downto 0))
else '0';
n11x <= n02x when to_reverse = '1' else n01x;
n11y <= n02y when to_reverse = '1' else n01y;
n11z <= n02z when to_reverse = '1' else n01z;
n21x <= n01x when to_reverse = '1' else n02x;
n21y <= n01y when to_reverse = '1' else n02y;
n21z <= n01z when to_reverse = '1' else n02z;
f2 <= theta2 when to_reverse = '1' else theta1;

-- v = cross(dp2p1, n1)
cross0_1x <= to_reverse & d2x(30 downto 0);
cross0_1y <= to_reverse & d2y(30 downto 0);
cross0_1z <= to_reverse & d2z(30 downto 0);
cross0_2x <= n1x;
cross0_2y <= n1y;
cross0_2z <= n1z;

-- v_norm = norm(v)
normal0_x <= cross0_resultx;
normal0_y <= cross0_resulty;
normal0_z <= cross0_resultz;

-- v[i] *= v_norm
mul2_x <= vx;
mul2_y <= normal0_result;
mul3_x <= vy;
mul3_y <= normal0_result;
mul4_x <= vz;
mul4_y <= normal0_result;

-- w = cross(n1, v)
cross1_1x <= n12x;
cross1_2x <= mul2_z;
cross1_1y <= n12y;
cross1_2y <= mul3_z;
cross1_1z <= n12z;
cross1_2z <= mul4_z;
wx <= cross1_resultx;
wy <= cross1_resulty;
wz <= cross1_resultz;

-- f[1] = dot(v, n2)
dot2_1x <= mul2_z;
dot2_2x <= n22x;
dot2_1y <= mul3_z;
dot2_2y <= n22y;
dot2_1z <= mul4_z;
dot2_2z <= n22z;
f1 <= dot2_result;

-- f[0] = atan2(temp1, temp2)
-- temp1 = dot(w, n2)
-- temp2 = dot(n1, n2)
dot3_1x <= wx;
dot3_2x <= n32x;
dot3_1y <= wy;
dot3_2y <= n32y;
dot3_1z <= wz;
dot3_2z <= n32z;
dot4_1x <= n11x;
dot4_2x <= n21x;
dot4_1y <= n11y;
dot4_2y <= n21y;
dot4_1z <= n11z;
dot4_2z <= n21z;
atan0_y <= dot3_result;
atan0_x <= delayed_atan0_y;
-- f0 <= atan0_result;

-- push outputs
feature0 <= atan0_result when is_zero = '0' else zero;
feature1 <= df1 when is_zero = '0' else zero;
feature2 <= df2 when is_zero = '0' else zero;

end behavioral;
B.10 Testbench

entity tb_pair_features is -- EDIT: name of testbench_file.
end tb_pair_features;

architecture tb of tb_pair_features is

-- Signals of TB here--
signal en : std_logic;
signal ptx : std_logic_vector(31 downto 0) := (others => '0');
signal pty : std_logic_vector(31 downto 0) := (others => '0');
signal ptz : std_logic_vector(31 downto 0) := (others => '0');
signal p2x : std_logic_vector(31 downto 0) := (others => '0');
signal p2y : std_logic_vector(31 downto 0) := (others => '0');
signal p2z : std_logic_vector(31 downto 0) := (others => '0');
signal n1x : std_logic_vector(31 downto 0) := (others => '0');
signal n1y : std_logic_vector(31 downto 0) := (others => '0');
signal n1z : std_logic_vector(31 downto 0) := (others => '0');
signal n2x : std_logic_vector(31 downto 0) := (others => '0');
signal n2y : std_logic_vector(31 downto 0) := (others => '0');
signal n2z : std_logic_vector(31 downto 0) := (others => '0');
signal f0 : std_logic_vector(31 downto 0);
signal f1 : std_logic_vector(31 downto 0);
signal f2 : std_logic_vector(31 downto 0);
signal f3 : std_logic_vector(31 downto 0);
signal drdy : std_logic_vector (0 downto 0);
signal valid : std_logic_vector (0 downto 0);
signal point_we : std_logic;
signal point_address_1 : std_logic_vector(9 downto 0);
signal point_address_2 : std_logic_vector(9 downto 0);
signal point_data_in : std_logic_vector(95 downto 0);
signal point_data_out_1 : std_logic_vector(95 downto 0);
signal point_data_out_2 : std_logic_vector(95 downto 0);
signal normal_we : std_logic;
signal normal_address_1 : std_logic_vector(9 downto 0);
signal normal_address_2 : std_logic_vector(9 downto 0);
signal normal_data_in : std_logic_vector(95 downto 0);
signal normal_data_out_1 : std_logic_vector(95 downto 0);
signal normal_data_out_2 : std_logic_vector(95 downto 0);

-----------------

--Clock signals--
-----------------
constant clock_period: time := 4 ns;
signal stop_the_clock: boolean;
signal clk: std_logic;

file point_file : text;
file normal_file : text;
file neighbor_file : text;
```vhdl
file file_out : text;
begin -- Architecture

tag : entity work.pair_features
port map(
    clk => clk,
    point1x => p1x,
    point1y => p1y,
    point1z => p1z,
    point2x => p2x,
    point2y => p2y,
    point2z => p2z,
    normal1x => n1x,
    normal1y => n1y,
    normal1z => n1z,
    normal2x => n2x,
    normal2y => n2y,
    normal2z => n2z,
    valid_inputs => valid,
    valid_outputs => drdy,
    feature0 => f0,
    feature1 => f1,
    feature2 => f2,
    feature3 => f3
);

point_ram : entity work.sim_ram
port map(
    clk => clk,
    we => point_we,
    address_1 => point_address_1,
    address_2 => point_address_2,
    data_in => point_data_in,
    data_out_1 => point_data_out_1,
    data_out_2 => point_data_out_2
);

normal_ram : entity work.sim_ram
port map(
    clk => clk,
    we => point_we,
    address_1 => normal_address_1,
    address_2 => normal_address_2,
    data_in => normal_data_in,
    data_out_1 => normal_data_out_1,
    data_out_2 => normal_data_out_2
);

-----------------------------
-- Clock setup --
-----------------------------
p1x <= point_data_out_1(95 downto 64);
p1y <= point_data_out_1(63 downto 32);
p1z <= point_data_out_1(31 downto 0);
```
p2x <= point_data_out_2(95 downto 64);
p2y <= point_data_out_2(63 downto 32);
p2z <= point_data_out_2(31 downto 0);

n1x <= normal_data_out_1(95 downto 64);
n1y <= normal_data_out_1(63 downto 32);
n1z <= normal_data_out_1(31 downto 0);
n2x <= normal_data_out_2(95 downto 64);
n2y <= normal_data_out_2(63 downto 32);
n2z <= normal_data_out_2(31 downto 0);

file_open(file_out, "scan_000_out.txt", write_mode);
file_open(neighbor_file, "scan_000_neighbors_hex.txt", read_mode);
file_open(point_file, "scan_000_cloud_hex.txt", read_mode);
file_open(normal_file, "scan_000_normals_hex.txt", read_mode);

clocking: process
begin
while not stop_the_clock loop
  clk <= '1', '0' after clock_period / 2;
  wait for clock_period;
end loop;
wait;
end process;

stimuli : process
variable v_ILINE : line;
variable v_OLINE : line;
variable v_px : std_ulogic_vector (31 downto 0);
variable v_py : std_ulogic_vector (31 downto 0);
variable v_pz : std_ulogic_vector (31 downto 0);
variable v_nx : std_ulogic_vector (31 downto 0);
variable v_ny : std_ulogic_vector (31 downto 0);
variable v_nz : std_ulogic_vector (31 downto 0);
variable v_k : std_ulogic_vector (11 downto 0);
variable address : integer range 0 to 1023;
variable v_SPACE : character;
variable test : std_logic_vector(31 downto 0);
begin -- Process stimuli
  -- Initialize all signal here --
  -- wait for 1 us;
  wait until falling_edge(clk);
  point_we <= '1';
  normal_we <= '1';
  address := 0;
  while not endfile(point_file) loop
    readline(point_file, v_ILINE);
    read(v_ILINE, v_px);
  -- Read values from file --
  readline(point_file, v_ILINE);
    read(v_ILINE, v_px);
read(v_ILINE, v_SPACE);
hread(v_ILINE, v_py);
read(v_ILINE, v_SPACE);
hread(v_ILINE, v_pz);

point_data_in <= std_logic_vector(v_px) & std_logic_vector(v_py) &
std_logic_vector(v_pz);
point_address_1 <= std_logic_vector(to_unsigned(address, 10));

readline(normal_file, v_ILINE);
hread(v_ILINE, v_nx);
read(v_ILINE, v_SPACE);
hread(v_ILINE, v_ny);
read(v_ILINE, v_SPACE);
hread(v_ILINE, v_nz);

normal_data_in <= std_logic_vector(v_nx) & std_logic_vector(v_ny) &
std_logic_vector(v_nz);
normal_address_1 <= std_logic_vector(to_unsigned(address, 10));

wait for clock_period;
address := address + 1;
end loop;

point_we <= '0';
normal_we <= '0';
address := 0;
valid <= "1";

while not endfile(neighbor_file) loop
-- for i in 0 to 5 loop
point_address_1 <= std_logic_vector(to_unsigned(address, 10));
normal_address_1 <= std_logic_vector(to_unsigned(address, 10));

readline(neighbor_file, v_ILINE);
hread(v_ILINE, v_k);
read(v_ILINE, v_SPACE);
while v_ILINE'length > 0 loop
hread(v_ILINE, v_k);
read(v_ILINE, v_SPACE);

point_address_2 <= std_logic_vector(v_k(9 downto 0));
normal_address_2 <= std_logic_vector(v_k(9 downto 0));
wait for clock_period;
end loop;
address := address + 1;
end loop;
valid <= "0";

wait for 500*clock_period;
stop_the_clock <= true;-- Stop the clock and hence terminate the simulation.
wait;
end process;
process(clk)
variable v_SPACE : character;
variable row : line;
begin
  if rising_edge(clk) and drdy = "1" then
    hwrite(row, f0, left, 10);
    hwrite(row, f1, left, 10);
    hwrite(row, f2, left, 10);
    hwrite(row, f3, left, 10);
    writeln(file_out, row);
  end if;
end process;
end tb;
B.11 C functions

/*
 * Reciprocal Square Root
 */
float rsr(float x){
    const float x2 = x * 0.5F;

    union {
        float f;
        uint32_t i;
    } conv = {x};
    conv.f = 0x5F375A86 - (conv.i >> 1);
    conv.f ^= (threehalves - (x2 * conv.f * conv.f));
    conv.f ^= (threehalves - (x2 * conv.f * conv.f));
    return conv.f;
    return 1/sqrt(x);
}

/*
 * Euclidean Distance
 */
float dist(vector<float> & point1, vector<float> & point2){
    float d = 0;
    float temp = 0;
    for (int i = 0; i < 3; i++){
        temp += pow((point1[i] - point2[i]),2);
    }
    d = sqrt(temp);
    return d;
}

/*
 * Reciprocal Euclidean Distance
 */
float rec_dist(vector<float> & point1, vector<float> & point2){
    float d = 0;
    float temp = 0;
    for (int i = 0; i < 3; i++){
        temp += pow((point1[i] - point2[i]),2);
    }
    d = rsr(temp);
    return d;
}

/*
 * Vector Magnitude
 */
float norm(vector<float> & vec){
    float n;
    float temp;
    for (int i = 0; i < vec.size(); i++){
        temp += pow(vec[i],2);
    }
    n = sqrt(temp);
}
```c
return n;
}
/
* Reciprocal Vector Magnitude */
float rec_norm(vector <float> & vec){
float n;
float temp;
for (int i = 0 ; i < vec.size(); i++){
    temp += pow(vec[i],2);
}n = rsr(temp);
return n;
}
/
* Dot Product */
float dot_product(vector <float>& vec1, vector <float>& vec2){
float dot = 0;
for (int i = 0; i < vec1.size(); i++){dot += vec1[i] * vec2[i];}
return dot;
}
/
* Cross Product */
void cross_product(vector <float>& vec1, vector <float>& vec2, vector <float>& cross){
}
/
* Atan Approximation */
float fast_atan(float x){float xx = x * x;return ((A*xx + B)*xx + C)*x;}
/
* Atan2 Approximation */
float my_atan2f(float y, float x){bool reversed = abs(x) < abs(y);
float temp_atan, result;
if (x == 0){if (y >0) return M_PI_2;
else if (y < 0) return -M_PI_2;
else return NAN;
}
```
```c
} else {
    if (reversed) temp_atan = fast_atan(x/y);
    else temp_atan = fast_atan(y/x);
    if (x > 0)
        result = temp_atan;
} else {
    if (y > 0) result = temp_atan + M_PI;
    else result = temp_atan - M_PI;
}
}
if (reversed) return M_PI_2 - result;
else return result;

} /*
* Original Pair Features
*/
void compute_pair_features(vector<float>& point1, vector<float>& point2, vector<float>& normal1, vector<float>& normal2, vector<float>& _features){
    _features[3] = dist(point1, point2);
    vector<float> dp2p1;
    dp2p1.resize(3,0);
    for (int i = 0 ; i < 3; i++)
        dp2p1[i] = point2[i] - point1[i];

    if (_features[3] == 0){
        _features[0] = 0;
        _features[1] = 0;
        _features[2] = 0;
        return;
    }

    vector<float> normal1_copy = normal1, normal2_copy = normal2;
    float angle1 = dot_product(normal1_copy, dp2p1)/_features[3];
    float angle2 = dot_product(normal2_copy, dp2p1)/_features[3];
    if (acos(fabs(angle1)) > acos(fabs(angle2))){
        normal1_copy.swap(normal2_copy);
        _features[2] = -angle2;
        for (int i = 0 ; i < 3; i++)
            dp2p1[i] *= (-1);
    }
    else{
        _features[2] = angle1;
    }

    //create Darboux frame coordinate system u, v, w (u=normal1)
    vector<float> v, w;
    v.resize(3,0);
    w.resize(3,0);
```
```c
B.11. C FUNCTIONS

cross_product(dp2p1, normal1_copy, v);
float v_norm = norm(v);
if (v_norm == 0){
    _features[0] = 0;
    _features[1] = 0;
    _features[2] = 0;
    _features[3] = 0;
}
for (int i = 0; i < 3; i++){
    v[i] /= v_norm;
}
cross_product(normal1_copy, v, w);
_features[1] = dot_product(v, normal2_copy);

// to help with fixed to float conversion
const float temp1, temp2;
const float temp3;
temp1 = dot_product(w, normal2_copy);
temp2 = dot_product(normal1_copy, normal2_copy);
temp3 = atan2f(temp1, temp2);
_features[0] = temp3;
}

/*
* Modified Pair Features
*/
void mod_compute_pair_features(vector<float>& point1, vector<float>& point2, vector<float>& normal1, vector<float>& normal2, vector<float>& _features){
    _features[3] = rec_dist(point1, point2);

    vector<float> dp2p1;
dp2p1.resize(3,0);
    for (int i = 0; i < 3; i++){
        dp2p1[i] = point2[i] - point1[i];
    }
    if (_features[3] == INFINITY){
        _features[0] = 0;
        _features[1] = 0;
        _features[2] = 0;
        return;
    }
    vector<float> normal1_copy = normal1, normal2_copy = normal2;
    float angle1 = dot_product(normal1_copy, dp2p1)*_features[3];
    float angle2 = dot_product(normal2_copy, dp2p1)*_features[3];
    if (fabs(angle1) < fabs(angle2)){
        normal1_copy.swap(normal2_copy);
        _features[2] = -angle2;
    }
    for (int i = 0; i < 3; i++){
        dp2p1[i] *= (-1);
    }
```
APPENDIX B. CODES

```cpp
} 

else{
    _features[2] = angle1;
}

//create Darboux frame coordinate system u, v, w (u= normal1)
vector <float> v, w;
v.resize(3.0);
w.resize(3.0);
cross_product(dp2p1, normal1_copy, v);
float v_norm = rec_norm(v);
if (v_norm == INFINITY){
    _features[0] = 0;
    _features[1] = 0;
    _features[2] = 0;
    _features[3] = 0;
}
for (int i = 0; i < 3; i++){
    v[i] *= v_norm;
}
cross_product(normal1_copy, v, w);

_features[1] = dot_product(v, normal2_copy);
// to help with fixed to float conversion
float temp1, temp2;
float temp3;
temp1 = dot_product(w, normal2_copy);
temp2 = dot_product(normal1_copy, normal2_copy);
temp3 = my_atan2f (temp1, temp2);
_features[0] = temp3;
}

/*
 * SPFH Calculation
 */
void compute_spfh(vector <vector <float>> & _points, vector <vector <float>> & _normals,
vector <vector <int>> & _neighbors,
vector <vector <float>> & _hist_f1, vector <vector <float>> & _hist_f2, vector <vector <
float>> & _hist_f3)
for (int i = 0; i < _neighbors.size(); i++){
    float hist_incr = 100.0 / (int) _neighbors[i].size();
    for (int j = 0; j < _neighbors[i].size(); j++){
        if (i == _neighbors[i][j]){ continue;
    }
    vector <float> features;
    features.resize(4.0);
    mod_compute_pair_features(_points[i], _points[_neighbors[i][j]], _normals[i], _normals[_neighbors[i][j]], features);
```
// Cout << features[0] << " \t" << features[1] << " \t" << features[2] << " \t" << features[3] << endl;

// Normalize features and push into histogram
int h_index = (int) (floor(bins * ((features[0] + PI) * d_pi)));
if (h_index < 0){
  h_index = 0;
}
if (h_index >= bins){
  h_index = bins - 1;
}
_hist_f1[i][h_index] += hist_incr;

h_index = (int) (floor(bins * ((features[1] + 1.0) * 0.5)));
if (h_index < 0){
  h_index = 0;
}
if (h_index >= bins){
  h_index = bins - 1;
}
_hist_f2[i][h_index] += hist_incr;

h_index = (int) (floor(bins * ((features[2] + 1.0) * 0.5)));
if (h_index < 0){
  h_index = 0;
}
if (h_index >= bins){
  h_index = bins - 1;
}
_hist_f3[i][h_index] += hist_incr;

*/

int main(int argc, char **argv){
  string filename = argv[1];
  //filename.erase(filename.end()-4, filename.end());
  string pointcloud = filename + "_cloud.txt";
  string normalscloud = filename + "_normals.txt";
  string neighborscloud = filename + "_neighbors.txt";
  string features_ref = filename + "_features_ref.txt";
  string features_test = filename + "_features_test.txt";

  // read data from pointcloud.txt
  cout << "Reading " << pointcloud << endl;
  vector<vector<float>> points;
  read_data(pointcloud, points);

  // read data from normals.txt
  cout << "Reading " << normalscloud << endl;
  vector<vector<float>> normals;

  /* Test to Compare Timings
   * Between Original and Modified Version */
read_data(normalscloud, normals);

// check if pointcloud and normalscloud have the same size
if (points.size() != normals.size())
    throw runtime_error("Pointcloud's size doesn't match normals' size.");

// for every point in pointcloud, read the closest neighbors
vector<vector<int>> neighbors;
for (int i = 0; i < pointcloud.size(); i++)
    read_data(neighborscloud, neighbors);

read_data(normalscloud, neighbors);
float total1 = 0, total2 = 0;
int cnt = 0;

vector<vector<float>> originals, custom;
int min_1, min_2, max_1, max_2;
max_1 = 0;
max_2 = 0;
min_1 = 5000;
min_2 = 5000;

for (int k = 0; k < 30; k++){
    for (int i = 0; i < neighbors.size(); i++)
        for (int j = 0; j < neighbors[i].size(); j++)
            if (i == neighbors[i][j])
                continue;

    vector<float> features_original, features_mod;
    features_original.resize(4, 0);
    features_mod.resize(4, 0);

    high_resolution_clock::time_point start_1 = high_resolution_clock::now();
    compute_pair_features(points[i], points[neighbors[i][j]], normals[i], normals[neighbors[i][j]], features_original);
    high_resolution_clock::time_point end_1 = high_resolution_clock::now();
    auto duration_1 = duration_cast<nanoseconds>(end_1 - start_1).count();

    high_resolution_clock::time_point start_2 = high_resolution_clock::now();
    mod_compute_pair_features(points[i], points[neighbors[i][j]], normals[i], normals[neighbors[i][j]], features_mod);
    high_resolution_clock::time_point end_2 = high_resolution_clock::now();
    auto duration_2 = duration_cast<nanoseconds>(end_2 - start_2).count();

    total1 += duration_1;
    total2 += duration_2;
    cnt ++;

    if (duration_1 > max_1){
        max_1 = duration_1;
    }
    if (duration_2 > max_2){
        max_2 = duration_2;
    }
    if (duration_1 < min_1){

min_1 = duration_1;
}
if (duration_2 < min_2){
    min_2 = duration_2;
}

originals.push_back(features_original);
custom.push_back(features_mod);

}}

}
Bibliography

Semantic 3d object maps for everyday manipulation in human living enviroments.
_Dissertaion._

Fast 3d recognition and pose using the viewpoint feature histogram.

Fast geometric point labeling using conditional random fields.

Fast point feature histograms (fpfh) for 3d registration.

Point cloud library online documentation.

Fast inverse square root.

Fast calculation of inverse square root with the use of magic constant – analytical approach.

A brief history of invsqrt.

What every computer scientis should know about floating-point arithmetic.

Floating point.