**INTERNATIONAL ORGANISATION FOR STANDARDISATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC1/SC29/WG11**

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC1/SC29/WG11 MPEG2019/m51011**

**October 2019, Geneva, CH**

|  |  |
| --- | --- |
| **Source** | **Apple Inc.** |
| **Status** | **Input document** |
| **Title** | **[G-PCC][New proposal] Using L1 norm for nearest neighbour search in Prediction and Lifting schemes.** |
| **Author** | **Zhenzhen Gao, David Flynn, Alexis Tourapis, and Khaled Mammou** |

# Abstract

The lifting and prediction schemes extensively use nearest neighbor searches during the predictors building stage. This process is highly parallelizable. In terms of HW implementation, multiple pipelines, each processing a group of points in parallel, could be used. The main limitation of such an approach is related to the chip area, especially that the considered L2 norm requires three multiplications and five additions to be applied for each norm computation. In this contribution we propose to replace the L2 norm for nearest neighbors’ computation by the L1 norm, which requires only five additions. The impact in terms of RD performance is about 0.3-1.2% loss in terms of Y/Cb/Cr -BD rate and 0.5-0.9% gain in terms of reflectance BD rate.

# Existing Implementation

In TMC13v7, nearest neighbors are determined based on an L2 norm. More precisely, let P(x0, y0, z0) be the considered point and Q(x1, y1, z1) a potential nearest neighbor. The square of the L2 distance between P and Q is computed as follows:

Computing the square of the L2 distance requires three multiplication operations and five additions. Furthermore, since the coordinates in G-PCC could use up to 21-bits, the results would require up to 44 bits to be stored.

# Proposed Implementation

We propose to use the L1 norm to determine the nearest neighbors of each point. The L1 distance between P and Q is defined as follows:

where is the absolute value of .

The L1 distance requires only five additions to be computed and could be stored on 24 bits, making it more HW friendly in terms of implementation

# Experimental results

The proposed approach was compared to the TMC13v7 anchor under the CTC conditions. Figure 4 shows that using L1 norm instead of the L2 norm has a limited impact of the RD performance:

* 0.3-1.2% loss in terms of average Y/Cb/Cr -BD rate, and
* 0.5-0.9% gain in terms of reflectance BD rate.

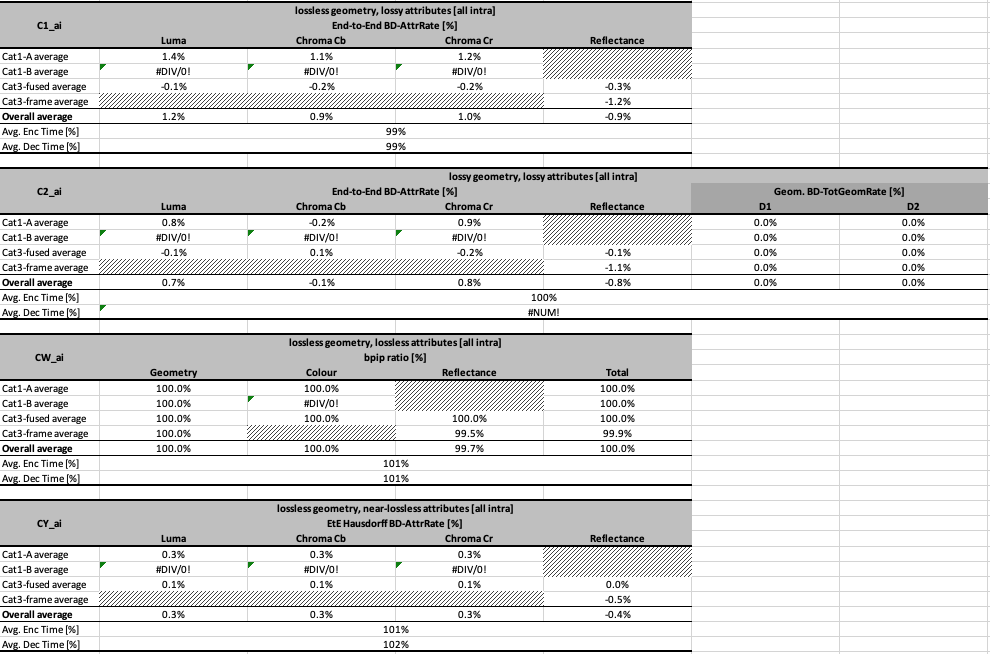


Figure 4. RD performance: proposed approach vs. TMC13v7.

# Specification change

In Section 8.3.22, replace the following

|  |
| --- |
| for (i = startIndex; i < endIndex; i++) {  currentIndex = assignedPointIndexes[i];  currentMortonCode = McodeUnsorted [currentIndex];  currentPos = PointPos[currentIndex];  while (j < nonAssignedPointCount &&  currentMortonCode >= McodeUnsorted[nonAssignedPointIndexes[j]) j++;  j = min(nonAssignedPointCount − 1, j);  j0 = max(0, j − searchRange);  j1 = min(nonAssignedPointCount, j + searchRange + 1);  neighboursCount[ currentIndex ] = 0;  k = 0;  for ( k = j0; k < j1 ; k++) {  neighbourIndex = nonAssignedPointIndex[k];  neighbourPos = PointPos[neighbourIndex];  d2 = (currentPos[0] – neighbourPos[ 0 ])2 + (currentPos[1] – neighbourPos[1]) 2 +  (currentPos[2] – neighbourPos[2]) 2;  insertIndex = ( k − j > 0 ) ? ( ( ( k – j ) << 1 ) – 1 ) : ( ( j – k ) << 1 );  If (neighboursCount[ currentIndex ] < numPredNearestNeighbours) {  p = neighboursCount[ currentIndex ];  neighbours[ currentIndex ][ p ] = neighbourIndex;  neighboursDistance2[ currentIndex ][ p ] = d2;  neighboursInsertIndex[ currentIndex ][ p ] = insertIndex;  neighboursCount[ currentIndex ]++;  sortNeighbours(neighboursCount[ currentIndex ], neighbours[ currentIndex ],  neighboursDistance2[ currentIndex ] ,  neighboursInsertIndex[ currentIndex ]);  } else if (d2 < neighboursDistance2[ currentIndex ] [ numPredNearestNeighbours−1) {  neighbours[ currentIndex ] [ numPredNearestNeighbours−1 = neighbourIndex;  neighboursDistance2[ currentIndex ] [ numPredNearestNeighbours−1 = d2;  neighboursInsertIndex[ currentIndex ][ numPredNearestNeighbours − 1] = insertIndex;  sortNeighbours(numPredNearestNeighbours, neighbours[ currentIndex ],  neighboursDistance2[ currentIndex ] ,  neighboursInsertIndex[ currentIndex ]);  }  }  if (currentLayer < liftingIntraLodPredictionNumLayers) {  j1 = min(endIndex, k + searchRange);  for ( k = i + 1; k < j1; k++) {  neighbourIndex = assignedPointIndex[k];  neighbourPos = PointPos[neighbourIndex];  d2 = (currentPos[0] – neighbourPos[ 0 ])2 + (currentPos[1] – neighbourPos[1]) 2 +  (currentPos[2] – neighbourPos[2]) 2;  insertIndex = ((k – i) << 1) – 1;  If (neighboursCount[ currentIndex ] < numPredNearestNeighbours) {  p = neighboursCount[ currentIndex ];  neighbours[ currentIndex ][ p ] = neighbourIndex;  neighboursDistance2[ currentIndex ][ p ] = d2;  neighboursInsertIndex[ currentIndex ][ p ] = insertIndex;  neighboursCount[ currentIndex ]++;  sortNeighbours(neighboursCount[ currentIndex ],  neighbours[ currentIndex ],  neighboursDistance2[ currentIndex ] ,  neighboursInsertIndex[ currentIndex ]);  } else if (d2 < neighboursDistance2[ currentIndex ][ numPredNearestNeighbours – 1]) {  neighbours[ currentIndex ][ numPredNearestNeighbours – 1 ] = neighbourIndex;  neighboursDistance2[ currentIndex ][ numPredNearestNeighbours – 1 ] = d2;  neighboursInsertIndex[ currentIndex ][ numPredNearestNeighbours − 1] = insertIndex;  sortNeighbours(numPredNearestNeighbours, neighbours[ currentIndex ],  neighboursDistance2[ currentIndex ],  neighboursInsertIndex[ currentIndex ]);  }  }  }  } |

With

|  |
| --- |
| for (i = startIndex; i < endIndex; i++) {  currentIndex = assignedPointIndexes[i];  currentMortonCode = McodeUnsorted [currentIndex];  currentPos = PointPos[currentIndex];  while (j < nonAssignedPointCount &&  currentMortonCode >= McodeUnsorted[nonAssignedPointIndexes[j]) j++;  j = min(nonAssignedPointCount − 1, j);  j0 = max(0, j − searchRange);  j1 = min(nonAssignedPointCount, j + searchRange + 1);  neighboursCount[ currentIndex ] = 0;  k = 0;  for ( k = j0; k < j1 ; k++) {  neighbourIndex = nonAssignedPointIndex[k];  neighbourPos = PointPos[neighbourIndex];  d2 = abs(currentPos[0] – neighbourPos[ 0 ]) + abs(currentPos[1] – neighbourPos[1]) +  abs(currentPos[2] – neighbourPos[2]);  insertIndex = ( k − j > 0 ) ? ( ( ( k – j ) << 1 ) – 1 ) : ( ( j – k ) << 1 );  If (neighboursCount[ currentIndex ] < numPredNearestNeighbours) {  p = neighboursCount[ currentIndex ];  neighbours[ currentIndex ][ p ] = neighbourIndex;  neighboursDistance2[ currentIndex ][ p ] = d2;  neighboursInsertIndex[ currentIndex ][ p ] = insertIndex;  neighboursCount[ currentIndex ]++;  sortNeighbours(neighboursCount[ currentIndex ], neighbours[ currentIndex ],  neighboursDistance2[ currentIndex ] ,  neighboursInsertIndex[ currentIndex ]);  } else if (d2 < neighboursDistance2[ currentIndex ] [ numPredNearestNeighbours−1) {  neighbours[ currentIndex ] [ numPredNearestNeighbours−1 = neighbourIndex;  neighboursDistance2[ currentIndex ] [ numPredNearestNeighbours−1 = d2;  neighboursInsertIndex[ currentIndex ][ numPredNearestNeighbours − 1] = insertIndex;  sortNeighbours(numPredNearestNeighbours, neighbours[ currentIndex ],  neighboursDistance2[ currentIndex ] ,  neighboursInsertIndex[ currentIndex ]);  }  }  if (currentLayer < liftingIntraLodPredictionNumLayers) {  j1 = min(endIndex, k + searchRange);  for ( k = i + 1; k < j1; k++) {  neighbourIndex = assignedPointIndex[k];  neighbourPos = PointPos[neighbourIndex];  d2 = abs(currentPos[0] – neighbourPos[ 0 ]) + abs(currentPos[1] – neighbourPos[1]) +  abs(currentPos[2] – neighbourPos[2]);  insertIndex = ((k – i) << 1) – 1;  If (neighboursCount[ currentIndex ] < numPredNearestNeighbours) {  p = neighboursCount[ currentIndex ];  neighbours[ currentIndex ][ p ] = neighbourIndex;  neighboursDistance2[ currentIndex ][ p ] = d2;  neighboursInsertIndex[ currentIndex ][ p ] = insertIndex;  neighboursCount[ currentIndex ]++;  sortNeighbours(neighboursCount[ currentIndex ],  neighbours[ currentIndex ],  neighboursDistance2[ currentIndex ] ,  neighboursInsertIndex[ currentIndex ]);  } else if (d2 < neighboursDistance2[ currentIndex ][ numPredNearestNeighbours – 1]) {  neighbours[ currentIndex ][ numPredNearestNeighbours – 1 ] = neighbourIndex;  neighboursDistance2[ currentIndex ][ numPredNearestNeighbours – 1 ] = d2;  neighboursInsertIndex[ currentIndex ][ numPredNearestNeighbours − 1] = insertIndex;  sortNeighbours(numPredNearestNeighbours, neighbours[ currentIndex ],  neighboursDistance2[ currentIndex ],  neighboursInsertIndex[ currentIndex ]);  }  }  }  } |

# Conclusion

In this contribution, we propose to replace the L2 norm for nearest neighbors’ computation by an L1 norm, which is more HW friendly in terms of HW implementation. The impact in terms of RD performance is about 0.3-1.2% loss in terms of Y/Cb/Cr -BD rate and 0.5-0.9% gain in terms of reflectance BD rate.

# References

1. “Common Test Conditions for PCC” ISO/IEC JTC1/SC29 WG11 MPEG2019 Doc. N18474, Geneva, CH, March 2019.