**INTERNATIONAL ORGANISATION FOR STANDARDISATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 11**

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC 1/SC 29/WG 11 M48918**

**Gothenburg, SE – July 2019**

|  |  |
| --- | --- |
| **Source:** | **Sony corporation** |
| **Title:** | **[G-PCC] (New) Lifting and RAHT harmonization** |
| **Authors:** | **Alexandre Zaghetto, Ali Tabatabai, Danillo Graziosi** |

**[G-PCC] (New) Lifting and RAHT harmonization**

# Introduction

In this document we propose five steps towards the unification of some aspects of Lifting and RAHT: (1) removal of the FixedPoint class in TMC13v6 [1]; (2) adoption of a unified quantization scheme, for both Lifting and RAHT transforms vs. the current practice of separate Lifting and RAHT quantization approaches. In addition, we also introduce a second LUT to remove the division operation in the forward quantization to follow similar practice done in AVC, HEVC and VVC; (3) as part of the unified quantization scheme we also allow additional deadzone and offset parameters to enable variations in offset (refers to *shiftFrac* in current TM13) and deadzone quantization, if so desired; (4) implementation of overflow handling in RAHT; and (5) increase the number of iterations of the approximation of the square root.

# Proposal

## Fixed-point class removal

The files FixedPoint.cpp and FixedPoint.h, that define the FixedPoint class, are removed and the RAHT fixed-point operations are performed directly in the code.

## LUT-based quantization

The general idea is to perform quantization using a forward quantization table and de-quantization using an inverse quantization table, as illustrated in the block diagram of Figure 1.

QP

fLUT

gLUT

Derive InvQstep

Derive ForQstep

De-quantization

Quantization

quantized

value

value

reconstructed value

Figure 1. Illustration of the proposed LUT-based quantization and

De-quantization.

Given a quantization parameter QP, a forward and an inverse quantization step are derived (ForQstep and InvQstep) based on two different quantization tables (forward *fLUT* and inverse *gLUT*). The input value is quantized using ForQStep and dequantized using InvQstep. The proposed scheme allows for the removal of the division operation in the quantization process, as will be described.

Consider the quantization table *gLUT*. This table is already used in TMC13v6 to perform forward and inverse quantization [2, 3]. Here, it will be used only as the inverse quantization table. The forward quantization table *fLUT* is derived as follows.

$$fLUT∙gLUT≈2^{M+N}\rightarrow fLUT= \frac{2^{M+N}}{gLUT}$$

Currently N = 8. And in our tests, we considered M = 14, since it was enough to achieve the desired precision. In this case, *fLUT* becomes,

## $$fLUT=\left[26052   23173   20662   18396   16384   14614\right].$$

## Fraction of the quantization step and Deadzone quantization

Regarding *PCCForQuantization*, an extra parameter (*shiftFrac*) was added to control the fraction of the quantization step used in the quantization procedure, as defined by the current *PCCQuantization* function. In the current implementation, *shiftFrac* is fixed and equal to 3 for Lifting. RAHT does not use the *PCCQuantization* function, but in its equivalent quantization procedure, *shiftFrac* is set to 2. We are proposing the use of the same value of *shiftFrac* = 3 for Lifting and RAHT. Deadzone quantization as adopted in the last meeting was also added as an option, but not used in our tests.

## Overflow handling

In the proposed implementation, instead of using an adjusted quantization step *aQs*, an adjusted coefficient *aCoef* is used. The equations for the adjusted quantization step and coefficient are given bellow.

$$aQs= \sqrt{\frac{Qs^{2}(wL+wR)}{wL×wR}},$$

$$aCoef= \sqrt{\frac{coef^{2}(wL×wR)}{wL+wR}}.$$

The multiplication $Qs^{2}\left(wL+wR\right)$ and $coef^{2}(wL×wR)$ may result in overflow. The current implementation of TMC13, which uses *aQs*, does not address overflow. In our implementation, which relies on *aCoef*, overflow handling is introduced.

We assume that the numerator of our fraction has a total of 64 bits available to represent $coef^{2}(wL×wR)$. First, the number of bits *Nc* need to represent $coef^{2}$ is estimated. Then, the number of bits *Na* available to represent $(wL×wR)$ is given by *Na* = 64-*Nc*. Let *Nw* be the actual number of bits needed to represent$ \left(wL×wR\right)$. If *Nc + Nw* exceeds 64, a shift of *S* = *Nw-Na* bits is applied to $\left(wL×wR\right)$and ($wL+wR)$. In this way, the overflow is avoided, while the value of the fraction $\left(wL×wR\right)/\left(wL+wR\right)$ is preserved, not affecting the final result.

A similar procedure is not adopted by the current implementation of RAHT. In ANNEX I we show a specific situation were overflow was detected. As one may deduce, it appears when the quantization step and the weights present large values.

## Square root

In the context of our implementation, a problem with the current approximation of the square root was observed. For lower numbers, the error in relation to the sqrt function of the C/C++ math library can be as big as 1%, which in some cases affects the result in the proposed RAHT adjusted coefficient/LUT-based quantization scheme. To avoid this problem, we propose to increase the number of iterations for the approximation of the square root, from 2 to 4. Consider the following simulation scenario:

for(uint64\_t i = 0; i<1000000000000/5; i+=5000000){

 x = isqrtCurrent(i)-sqrt((double)i);

 y = isqrtProposed-sqrt((double)i);

}

The plots on the left and on the right represent errors for the current and the proposed solution, respectively.

 

If, for instance, and error occurs in the quantization of a DC coefficient, which involves the computation of a square root, the distortion may become significant.

# Results

In all experiments we used the following quantization table:

* $gLUT=[161   181   203   228   256   287] $;
* $fLUT=\left[26052   23173   20662   18396   16384   14614\right];$

Next, results for different scenarios are shown for RAHT and Lifting.

## RAHT

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Experiment | *Transform* | *shiftFrac* | square root iterations | coefficient overflow handling |
| 1 | RAHT | 2 | 2 | False |
| 2 | RAHT | 3 | 2 | False |
| 3 | RAHT | 3 | 2 | True |
| 4 | RAHT | 3 | 4 | True |
| 5 | Lifting | 3 | 4 | Not applicable |

Results for Experiments 1, 2 and 3 were generated for C1, cat1-A. For Experiment 4, cat1-B and cat3-fused were added, as well as C2, cat1-A, cat1-B and cat3-fused. Results for Lifting are included only to show that they are not significantly affected by the change in the square root approximation. Details for all results are in the attached excel files.

### Experiment 1

From the detailed excel files one may notice that mostly the results match the anchor. The errors are localized, but when they occur, they are significantly large and compromise the overall performance.



### Experiment 2

By changing *shiftFrac* from 2 to 3, significant gains over anchor are reported for most of the dataset. However, the points where the mismatch was large, the error still remains.



### Experiment 3

As overflow handling was introduced, almost all points outperformed anchor’s results, but some isolated points still presented large mismatched results.

###

### Experiment 4 (best configuration)

With the addition of 2 more iterations to the square root approximation, the proposed scheme outperformed the anchor in all tested cases, as shown in the table below.



## LIFTING

### Experiment 5

This experiment shows that the adoption of the LUT-based quantization and increase of the number of iterations in the approximation of the square root did not affect Lifting significantly.



# Conclusion

In this document, we presented a proposal of unification of some aspects of Lifting and RAHT. For the reduced test set described in Section 3, the presented proposal outperform the anchor TMC13v6, showing that the solution is viable. One may notice that there is a gradual increase of performance when: (1) *shiftFrac* changes from 2 to 3; (2) overflow handling is introduced; and (3) the number of iterations of square roots approximation is increased from 2 to 4. For Lifting, the differences in relation to the anchor are negligible and are a consequence of a better approximation of the square root. For RAHT, average results are around 8% to 12% better then Anchor. For a better understanding of the influence of the parameter used in the provided Experiments, consult the excel files attached to this document.

# Reference

1. “G-PCC TMC13v6”, ISO/IEC JTC1/SC29/WG11 MPEG2019 Doc. w18476, Geneva, Switzerland, March 2019.
2. “G-PCC On an improvement of RAHT to exploit attribute correlation”, Doc. m47378, Geneva, Switzerland, March 2019.
3. Ali Tabatabai, Alexandre Zaghetto, Danillo Graziosi, “[G-PCC] New contribution on quantization parameter definition,” Geneva, Switzerland, March 2019.
4. Noritaka Iguchi, Chung Dean Han, “[G-PCC] Quantization Parameter table in Attribute Coding,” Geneva, Switzerland, March 2019.

**ANNEX I – Overflow in TMC13v6.0**

In TMC13v6.0 [4] the definition of the quantization step as a function of the quantization parameter was adopted [5]. Compared to TMC13v5.0 the new version introduced an additional multiplication factor of 2562 in the computation of the a adjustedQuantStepSize.val, as will be exemplified next.

Let’s assume sliceQpLuma = 52 (r01), then

qpShiftLuma = sliceQpLuma / 6 = 8

qstep = kQpStep[sliceQpLuma % 6] << qpShiftLuma 🡺 kQpStep[4] << 8.

Since

kQpStep[6] = {161, 181, 203, 228, 256, 287},

qstep = 256 \* 256 = 65536.

However, the constructor of the FixedPoint class scales qstep[0] by 1<<kFracBits = 256.

In this example, quantStepSizeLuma becomes 65536\*256 = 16777216.





And in rahtFixedPointRotation the computation of the adjustedQuantStepSize.val results in overflow.



This is one example where overflow can be observed for Egyptian\_mask\_vox12, octree-raht, C1: losslG-lossyA, just before the quantization of attributeTransformedHigh = 4206. The multiplication of (quantStepSizeLuma.val \* quantStepSizeLuma.val) \* (weightLeft + weightRight) yields a number that exceeds the uint64\_t upper bound. The arrow points out the overflow. The real value should be 3.8211354e+19.

