原 FFmpeg的HEVC解码器源代码简单分析：CTU解码（CTU Decode）部分-PU

2015年06月15日 17:32:50　阅读数：5317

==========

HEVC源代码分析文章列表：

【解码 -libavcodec HEVC 解码器】

FFmpeg的HEVC解码器源代码简单分析：概述

FFmpeg的HEVC解码器源代码简单分析：解析器（Parser）部分

FFmpeg的HEVC解码器源代码简单分析：解码器主干部分

FFmpeg的HEVC解码器源代码简单分析：CTU解码（CTU Decode）部分-PU

FFmpeg的HEVC解码器源代码简单分析：CTU解码（CTU Decode）部分-TU

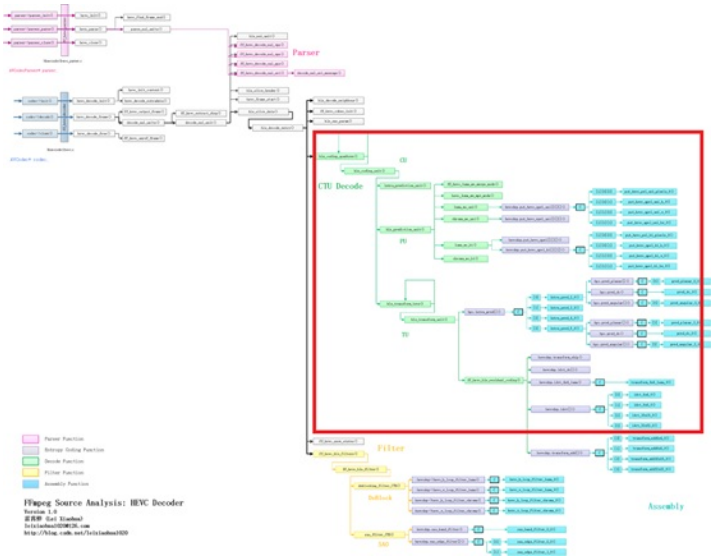
FFmpeg的HEVC解码器源代码简单分析：环路滤波（LoopFilter）

==========

本文分析FFmpeg的libavcodec中的HEVC解码器的CTU解码（CTU Decode）部分的源代码。FFmpeg的HEVC解码器调用hls_decode_entry()函数完成了Slice解码工作。hls_decode_entry()则调用了hls_coding_quadtree()完成了CTU解码工作。由于CTU解码部分的内容比较多，因此将这一部分内容拆分成两篇文章：一篇文章记录PU的解码，另一篇文章记录TU解码。本文记录PU的解码过程。

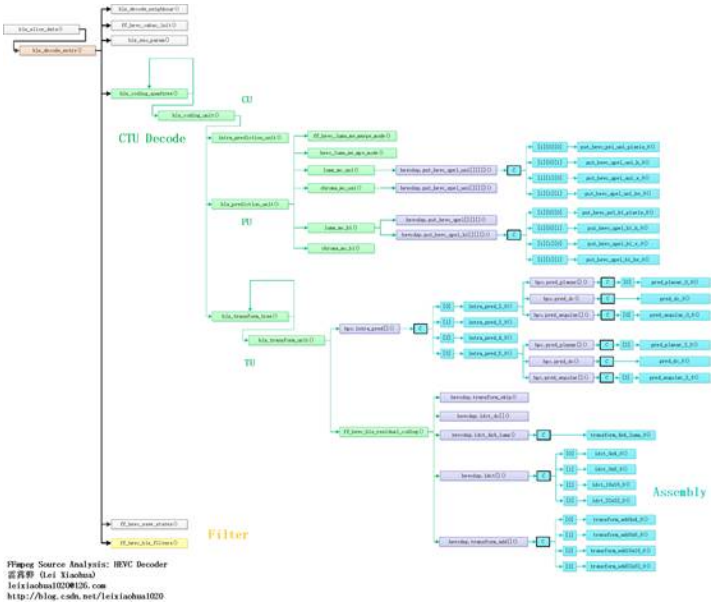
函数调用关系图

FFmpeg HEVC解码器的CTU解码（CTU Decoder）部分在整个HEVC解码器中的位置如下图所示。



单击查看更清晰的大图

CTU解码（CTU Decoder）部分的函数调用关系如下图所示。



单击查看更清晰的大图

从图中可以看出，CTU解码模块对应的函数是hls_coding_quadtree()。该函数是一个递归调用的函数，可以按照四叉树的句法格式解析CTU并获得其中的CU。对于每个CU会调用hls_coding_unit()进行解码。

hls_coding_unit()会调用hls_prediction_unit()对CU中的PU进行处理。hls_prediction_unit()调用luma_mc_uni()对亮度单向预测块进行运动补偿处理，调用chroma_mc_uni()对色度单向预测块进行运动补偿处理，调用luma_mc_bi()对亮度单向预测块进行运动补偿处理。

hls_coding_unit()会调用hls_transform_tree()对CU中的TU进行处理。hls_transform_tree()是一个递归调用的函数，可以按照四叉树的句法格式解析并获得其中的TU。对于每一个TU会调用hls_transform_unit()进行解码。hls_transform_unit()会进行帧内预测，并且调用ff_hevc_hls_residual_coding()解码DCT残差数据。

hls_decode_entry()

hls_decode_entry()是FFmpeg HEVC解码器中Slice解码的入口函数。该函数的定义如下所示。

```cpp
//解码入口函数
static int hls_decode_entry(AVCodecContext *avctxt, void *isFilterThread)
{
    HEVCContext *s  = avctxt->priv_data;
    //CTB尺寸
    int ctb_size    = 1 << s->sps->log2_ctb_size;
    int more_data   = 1;
    int x_ctb       = 0;
    int y_ctb       = 0;
    int ctb_addr_ts = s->pps->ctb_addr_rs_to_ts[s->sh.slice_ctb_addr_rs];

    if (!ctb_addr_ts && s->sh.dependent_slice_segment_flag) {
        av_log(s->avctx, AV_LOG_ERROR, "Impossible initial tile.\n");
        return AVERROR_INVALIDDATA;
    }

    if (s->sh.dependent_slice_segment_flag) {
        int prev_rs = s->pps->ctb_addr_ts_to_rs[ctb_addr_ts - 1];
        if (s->tab_slice_address[prev_rs] != s->sh.slice_addr) {
            av_log(s->avctx, AV_LOG_ERROR, "Previous slice segment missing\n");
            return AVERROR_INVALIDDATA;
        }
    }

    while (more_data && ctb_addr_ts < s->sps->ctb_size) {
        int ctb_addr_rs = s->pps->ctb_addr_ts_to_rs[ctb_addr_ts];
        //CTB的位置x和y
        x_ctb = (ctb_addr_rs % ((s->sps->width + ctb_size - 1) >> s->sps->log2_ctb_size)) << s->sps->log2_ctb_size;
        y_ctb = (ctb_addr_rs / ((s->sps->width + ctb_size - 1) >> s->sps->log2_ctb_size)) << s->sps->log2_ctb_size;
        //初始化周围的参数
        hls_decode_neighbour(s, x_ctb, y_ctb, ctb_addr_ts);
        //初始化CABAC
        ff_hevc_cabac_init(s, ctb_addr_ts);
        //样点自适应补偿参数
        hls_sao_param(s, x_ctb >> s->sps->log2_ctb_size, y_ctb >> s->sps->log2_ctb_size);

        s->deblock[ctb_addr_rs].beta_offset = s->sh.beta_offset;
        s->deblock[ctb_addr_rs].tc_offset   = s->sh.tc_offset;
        s->filter_slice_edges[ctb_addr_rs]  = s->sh.slice_loop_filter_across_slices_enabled_flag;
        /*
         * CU示意图
         *
```

```
         * 64x64块
         *
         * 深度d=0
         * split_flag=1时候划分为4个32x32
         *
         * +--------+--------+--------+--------+--------+--------+--------+--------+
         * |                                                                       |
         * |                                  |                                    |
         * |                                                                       |
         * +                                  |                                    +
         * |                                                                       |
         * |                                  |                                    |
         * |                                                                       |
         * +                                  |                                    +
         * |                                                                       |
         * |                                  |                                    |
         * |                                                                       |
         * +                                  |                                    +
         * |                                                                       |
         * |                                  |                                    |
         * |                                                                       |
         * + --  --  --  --  --  --  --  --  --+ --  --  --  --  --  --  --  --  --+
         * |                                  |                                    |
         * |                                                                       |
         * |                                  |                                    |
         * +                                                                       +
         * |                                  |                                    |
         * |                                                                       |
         * |                                  |                                    |
         * +                                                                       +
         * |                                  |                                    |
         * |                                                                       |
         * |                                  |                                    |
         * +                                                                       +
         * |                                  |                                    |
         * |                                                                       |
         * |                                  |                                    |
         * +--------+--------+--------+--------+--------+--------+--------+--------+
         *
         *
         * 32x32 块
         * 深度d=1
         * split_flag=1时候划分为4个16x16
         *
         * +--------+--------+--------+--------+
         * |                                  |
         * |                |                 |
         * |                                  |
         * +                |                 +
         * |                                  |
         * |                |                 |
         * |                                  |
         * + --  --  --  --  + --  --  --  --  +
         * |                                  |
         * |                |                 |
         * |                                  |
         * +                |                 +
         * |                                  |
         * |                |                 |
         * |                                  |
         * +--------+--------+--------+--------+
         *
         *
         * 16x16 块
         * 深度d=2
         * split_flag=1时候划分为4个8x8
         *
         * +--------+--------+
         * |                 |
         * |        |        |
         * |                 |
         * +  --  --+ --  -- +
         * |                 |
         * |        |        |
         * |                 |
         * +--------+--------+
         *
         *
         * 8x8块
         * 深度d=3
         * split_flag=1时候划分为4个4x4
         *
         * +----+----+
         * |    |    |
         * + -- + -- +
         * |    |    |
         * +----+----+
         *
         */
        /*
         * 解析四叉树结构，并且解码
```

```
         *
         * hls_coding_quadtree(HEVCContext *s, int x0, int y0, int log2_cb_size, int cb_depth)中：
         * s：HEVCContext上下文结构体
         * x_ctb：CB位置的x坐标
         * y_ctb：CB位置的y坐标
         * log2_cb_size：CB大小取log2之后的值
         * cb_depth：深度
         *
         */
        more_data = hls_coding_quadtree(s, x_ctb, y_ctb, s->sps->log2_ctb_size, 0);
        if (more_data < 0) {
            s->tab_slice_address[ctb_addr_rs] = -1;
            return more_data;
        }


        ctb_addr_ts++;
        //保存解码信息以供下次使用
        ff_hevc_save_states(s, ctb_addr_ts);
        //去块效应滤波
        ff_hevc_hls_filters(s, x_ctb, y_ctb, ctb_size);
    }

    if (x_ctb + ctb_size >= s->sps->width &&
        y_ctb + ctb_size >= s->sps->height)
        ff_hevc_hls_filter(s, x_ctb, y_ctb, ctb_size);

    return ctb_addr_ts;
}
```

从源代码可以看出，hls_decode_entry()主要调用了2个函数进行解码工作：

（1）调用hls_coding_quadtree()解码CTU。其中包含了PU和TU的解码。

（2）调用ff_hevc_hls_filters()进行滤波。其中包含了去块效应滤波和SAO滤波。

本文分析第一步的PU解码过程。

hls_coding_quadtree()

hls_coding_quadtree()用于解析CTU的四叉树句法结构。该函数的定义如下所示。

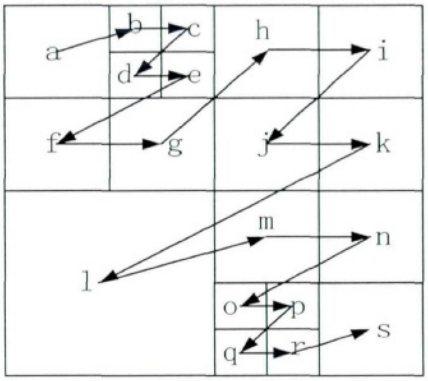
```cpp
/*
 * 解析四叉树结构，并且解码
 * 注意该函数是递归调用
 * 注释和处理：雷霄骅
 *
 *
 * s：HEVCContext上下文结构体
 * x_ctb：CB位置的x坐标
 * y_ctb：CB位置的y坐标
 * log2_cb_size：CB大小取log2之后的值
 * cb_depth：深度
 *
 */
static int hls_coding_quadtree(HEVCContext *s, int x0, int y0,
                               int log2_cb_size, int cb_depth)
{
    HEVCLocalContext *lc = s->HEVClc;
    //CB的大小,split flag=0
    //log2_cb_size为CB大小取log之后的结果
    const int cb_size    = 1 << log2_cb_size;
    int ret;
    int qp_block_mask = (1<<(s->sps->log2_ctb_size - s->pps->diff_cu_qp_delta_depth)) - 1;
    int split_cu;
    //确定CU是否还会划分？
    lc->ct_depth = cb_depth;
    if (x0 + cb_size <= s->sps->width  &&
        y0 + cb_size <= s->sps->height &&
        log2_cb_size > s->sps->log2_min_cb_size) {
        split_cu = ff_hevc_split_coding_unit_flag_decode(s, cb_depth, x0, y0);
    } else {
        split_cu = (log2_cb_size > s->sps->log2_min_cb_size);
    }
    if (s->pps->cu_qp_delta_enabled_flag &&
        log2_cb_size >= s->sps->log2_ctb_size - s->pps->diff_cu_qp_delta_depth) {
        lc->tu.is_cu_qp_delta_coded = 0;
        lc->tu.cu_qp_delta          = 0;
    }

    if (s->sh.cu_chroma_qp_offset_enabled_flag &&
        log2_cb_size >= s->sps->log2_ctb_size - s->pps->diff_cu_chroma_qp_offset_depth) {
        lc->tu.is_cu_chroma_qp_offset_coded = 0;
    }

    if (split_cu) {
        //如果CU还可以继续划分，则继续解析划分后的CU
        //注意这里是递归调用
```

```
47.
48.
49.         //CB的大小,split flag=1
50.         const int cb_size_split = cb_size >> 1;
51.
52.         /*
53.          * (x0, y0)  (x1, y0)
54.          *     +--------+--------+
55.          *     |                 |
56.          *     |        |        |
57.          *     |                 |
58.          *     +  --  --+ --  -- +
59.          * (x0, y1)  (x1, y1)    |
60.          *     |        |        |
61.          *     |                 |
62.          *     +--------+--------+
63.          *
64.          */
65.         const int x1 = x0 + cb_size_split;
66.         const int y1 = y0 + cb_size_split;
67.
68.         int more_data = 0;
69.
70.         //注意：
71.         //CU大小减半，log2_cb_size-1
72.         //深度d加1，cb_depth+1
73.         more_data = hls_coding_quadtree(s, x0, y0, log2_cb_size - 1, cb_depth + 1);
74.         if (more_data < 0)
75.             return more_data;
76.
77.         if (more_data && x1 < s->sps->width) {
78.             more_data = hls_coding_quadtree(s, x1, y0, log2_cb_size - 1, cb_depth + 1);
79.             if (more_data < 0)
80.                 return more_data;
81.         }
82.         if (more_data && y1 < s->sps->height) {
83.             more_data = hls_coding_quadtree(s, x0, y1, log2_cb_size - 1, cb_depth + 1);
84.             if (more_data < 0)
85.                 return more_data;
86.         }
87.         if (more_data && x1 < s->sps->width &&
88.             y1 < s->sps->height) {
89.             more_data = hls_coding_quadtree(s, x1, y1, log2_cb_size - 1, cb_depth + 1);
90.             if (more_data < 0)
91.                 return more_data;
92.         }
93.
94.         if(((x0 + (1<<log2_cb_size)) & qp_block_mask) == 0 &&
95.             ((y0 + (1<<log2_cb_size)) & qp_block_mask) == 0)
96.             lc->qPy_pred = lc->qp_y;
97.
98.         if (more_data)
99.             return ((x1 + cb_size_split) < s->sps->width ||
100.                     (y1 + cb_size_split) < s->sps->height);
101.         else
102.             return 0;
103.     } else {
104.
105.         /*
106.          * (x0, y0)
107.          *     +--------+--------+
108.          *     |                 |
109.          *     |                 |
110.          *     |                 |
111.          *     +                 +
112.          *     |                 |
113.          *     |                 |
114.          *     |                 |
115.          *     +--------+--------+
116.          *
117.          */
118.         //注意处理的是不可划分的CU单元
119.         //处理CU单元-真正的解码
120.         ret = hls_coding_unit(s, x0, y0, log2_cb_size);
121.         if (ret < 0)
122.             return ret;
123.         if ((!((x0 + cb_size) %
124.                (1 << (s->sps->log2_ctb_size))) ||
125.              (x0 + cb_size >= s->sps->width)) &&
126.             (!((y0 + cb_size) %
127.                (1 << (s->sps->log2_ctb_size))) ||
128.              (y0 + cb_size >= s->sps->height))) {
129.             int end_of_slice_flag = ff_hevc_end_of_slice_flag_decode(s);
130.             return !end_of_slice_flag;
131.         } else {
132.             return 1;
133.         }
134.     }
135.
136.     return 0;
137. }
```

从源代码可以看出，hls_coding_quadtree()首先调用ff_hevc_split_coding_unit_flag_decode()判断当前CU是否还需要划分。如果需要划分的话，就会递归调用4次hls_coding_quadtree()分别对4个子块继续进行四叉树解析；如果不需要划分，就会调用hls_coding_unit()对CU进行解码。总而言之，hls_coding_quadtree()会解析出来一个CTU中的所有CU，并且对每一个CU逐一调用hls_coding_unit()进行解码。一个CTU中CU的解码顺序如下图所示。图中a, b, c …即代表了的先后顺序。



hls_coding_unit()

hls_coding_unit()用于解码一个CU。该函数的定义如下所示。

```cpp
//处理CU单元-真正的解码
static int hls_coding_unit(HEVCContext *s, int x0, int y0, int log2_cb_size)
{
    //CB大小
    int cb_size          = 1 << log2_cb_size;
    HEVCLocalContext *lc = s->HEVClc;
    int log2_min_cb_size = s->sps->log2_min_cb_size;
    int length           = cb_size >> log2_min_cb_size;
    int min_cb_width     = s->sps->min_cb_width;
    //以最小的CB为单位（例如4x4）的时候，当前CB的位置—x坐标和y坐标
    int x_cb             = x0 >> log2_min_cb_size;
    int y_cb             = y0 >> log2_min_cb_size;
    int idx              = log2_cb_size - 2;
    int qp_block_mask    = (1<<(s->sps->log2_ctb_size - s->pps->diff_cu_qp_delta_depth)) - 1;
    int x, y, ret;

    //设置CU的属性值
    lc->cu.x                = x0;
    lc->cu.y                = y0;
    lc->cu.pred_mode        = MODE_INTRA;
    lc->cu.part_mode        = PART_2Nx2N;
    lc->cu.intra_split_flag = 0;

    SAMPLE_CTB(s->skip_flag, x_cb, y_cb) = 0;

    for (x = 0; x < 4; x++)
        lc->pu.intra_pred_mode[x] = 1;
    if (s->pps->transquant_bypass_enable_flag) {
        lc->cu.cu_transquant_bypass_flag = ff_hevc_cu_transquant_bypass_flag_decode(s);
        if (lc->cu.cu_transquant_bypass_flag)
            set_deblocking_bypass(s, x0, y0, log2_cb_size);
    } else
        lc->cu.cu_transquant_bypass_flag = 0;

    if (s->sh.slice_type != I_SLICE) {
        //Skip类型
        uint8_t skip_flag = ff_hevc_skip_flag_decode(s, x0, y0, x_cb, y_cb);
        //设置到skip_flag缓存中
        x = y_cb * min_cb_width + x_cb;
        for (y = 0; y < length; y++) {
            memset(&s->skip_flag[x], skip_flag, length);
            x += min_cb_width;
        }
        lc->cu.pred_mode = skip_flag ? MODE_SKIP : MODE_INTER;
    } else {
        x = y_cb * min_cb_width + x_cb;
        for (y = 0; y < length; y++) {
            memset(&s->skip_flag[x], 0, length);
            x += min_cb_width;
        }
    }

    if (SAMPLE_CTB(s->skip_flag, x_cb, y_cb)) {
        hls_prediction_unit(s, x0, y0, cb_size, cb_size, log2_cb_size, 0, idx);
        intra_prediction_unit_default_value(s, x0, y0, log2_cb_size);

        if (!s->sh.disable_deblocking_filter_flag)
```

```
            ff_hevc_deblocking_boundary_strengths(s, x0, y0, log2_cb_size);
    } else {
        int pcm_flag = 0;

        //读取预测模式（非 I Slice）
        if (s->sh.slice_type != I_SLICE)
            lc->cu.pred_mode = ff_hevc_pred_mode_decode(s);

        //不是帧内预测模式的时候
        //或者已经是最小CB的时候
        if (lc->cu.pred_mode != MODE_INTRA ||
            log2_cb_size == s->sps->log2_min_cb_size) {
            //读取CU分割模式
            lc->cu.part_mode        = ff_hevc_part_mode_decode(s, log2_cb_size);
            lc->cu.intra_split_flag = lc->cu.part_mode == PART_NxN &&
                                      lc->cu.pred_mode == MODE_INTRA;
        }

        if (lc->cu.pred_mode == MODE_INTRA) {
            //帧内预测模式

            //PCM方式编码，不常见
            if (lc->cu.part_mode == PART_2Nx2N && s->sps->pcm_enabled_flag &&
                log2_cb_size >= s->sps->pcm.log2_min_pcm_cb_size &&
                log2_cb_size <= s->sps->pcm.log2_max_pcm_cb_size) {
                pcm_flag = ff_hevc_pcm_flag_decode(s);
            }
            if (pcm_flag) {
                intra_prediction_unit_default_value(s, x0, y0, log2_cb_size);
                ret = hls_pcm_sample(s, x0, y0, log2_cb_size);
                if (s->sps->pcm.loop_filter_disable_flag)
                    set_deblocking_bypass(s, x0, y0, log2_cb_size);

                if (ret < 0)
                    return ret;
            } else {
                //获取帧内预测模式
                intra_prediction_unit(s, x0, y0, log2_cb_size);
            }
        } else {
            //帧间预测模式
            intra_prediction_unit_default_value(s, x0, y0, log2_cb_size);

            //帧间模式一共有8种划分模式

            switch (lc->cu.part_mode) {
            case PART_2Nx2N:
                /*
                 * PART_2Nx2N:
                 * +--------+--------+
                 * |                 |
                 * |                 |
                 * |                 |
                 * +        +        +
                 * |                 |
                 * |                 |
                 * |                 |
                 * +--------+--------+
                 */
                //处理PU单元-运动补偿
                hls_prediction_unit(s, x0, y0, cb_size, cb_size, log2_cb_size, 0, idx);
                break;
            case PART_2NxN:
                /*
                 * PART_2NxN:
                 * +--------+--------+
                 * |                 |
                 * |                 |
                 * |                 |
                 * +--------+--------+
                 * |                 |
                 * |                 |
                 * |                 |
                 * +--------+--------+
                 *
                 */
                /*
                 * hls_prediction_unit()参数：
                 * x0 : PU左上角x坐标
                 * y0 : PU左上角y坐标
                 * nPbW : PU宽度
                 * nPbH : PU高度
                 * log2_cb_size : CB大小取log2()的值
                 * partIdx : PU的索引号-分成4个块的时候取0-3，分成两个块的时候取0和1
                 */
                //上
                hls_prediction_unit(s, x0, y0,               cb_size, cb_size / 2, log2_cb_size, 0, idx);
                //下
                hls_prediction_unit(s, x0, y0 + cb_size / 2, cb_size, cb_size / 2, log2_cb_size, 1, idx);
                break;
            case PART_Nx2N:
```

```
                /*
                 * PART_Nx2N:
                 * +--------+--------+
                 * |        |        |
                 * |        |        |
                 * |        |        |
                 * +        +        +
                 * |        |        |
                 * |        |        |
                 * |        |        |
                 * +--------+--------+
                 *
                 */
                //左
                hls_prediction_unit(s, x0,               y0, cb_size / 2, cb_size, log2_cb_size, 0, idx - 1);
                //右
                hls_prediction_unit(s, x0 + cb_size / 2, y0, cb_size / 2, cb_size, log2_cb_size, 1, idx - 1);
                break;
            case PART_2NxnU:
                /*
                 * PART_2NxnU (Upper) :
                 * +--------+--------+
                 * |                 |
                 * +--------+--------+
                 * |                 |
                 * +        +        +
                 * |                 |
                 * |                 |
                 * |                 |
                 * +--------+--------+
                 *
                 */
                //上
                hls_prediction_unit(s, x0, y0,               cb_size, cb_size     / 4, log2_cb_size, 0, idx);
                //下
                hls_prediction_unit(s, x0, y0 + cb_size / 4, cb_size, cb_size * 3 / 4, log2_cb_size, 1, idx);
                break;
            case PART_2NxnD:
                /*
                 * PART_2NxnD (Down) :
                 * +--------+--------+
                 * |                 |
                 * |                 |
                 * |                 |
                 * +        +        +
                 * |                 |
                 * +--------+--------+
                 * |                 |
                 * +--------+--------+
                 *
                 */
                //上
                hls_prediction_unit(s, x0, y0,                   cb_size, cb_size * 3 / 4, log2_cb_size, 0, idx);
                //下
                hls_prediction_unit(s, x0, y0 + cb_size * 3 / 4, cb_size, cb_size     / 4, log2_cb_size, 1, idx);
                break;
            case PART_nLx2N:
                /*
                 * PART_nLx2N (Left):
                 * +----+---+--------+
                 * |    |            |
                 * |    |            |
                 * |    |            |
                 * +    +   +        +
                 * |    |            |
                 * |    |            |
                 * |    |            |
                 * +----+---+--------+
                 *
                 */
                //左
                hls_prediction_unit(s, x0,               y0, cb_size     / 4, cb_size, log2_cb_size, 0, idx - 2);
                //右
                hls_prediction_unit(s, x0 + cb_size / 4, y0, cb_size * 3 / 4, cb_size, log2_cb_size, 1, idx - 2);
                break;
            case PART_nRx2N:
                /*
                 * PART_nRx2N (Right):
                 * +--------+---+----+
                 * |            |    |
                 * |            |    |
                 * |            |    |
                 * +        +   +    +
                 * |            |    |
                 * |            |    |
                 * |            |    |
                 * +--------+---+----+
                 *
                 */
                //左
                hls_prediction_unit(s, x0,                   y0, cb_size * 3 / 4, cb_size, log2_cb_size, 0, idx - 2);
```

```
                //右
                hls_prediction_unit(s, x0 + cb_size * 3 / 4, y0, cb_size     / 4, cb_size, log2_cb_size, 1, idx - 2);
                break;
            case PART_NxN:
                /*
                 * PART_NxN:
                 * +--------+--------+
                 * |        |        |
                 * |        |        |
                 * |        |        |
                 * +--------+--------+
                 * |        |        |
                 * |        |        |
                 * |        |        |
                 * +--------+--------+
                 *
                 */
                hls_prediction_unit(s, x0,               y0,               cb_size / 2, cb_size / 2, log2_cb_size, 0, idx - 1);
                hls_prediction_unit(s, x0 + cb_size / 2, y0,               cb_size / 2, cb_size / 2, log2_cb_size, 1, idx - 1);
                hls_prediction_unit(s, x0,               y0 + cb_size / 2, cb_size / 2, cb_size / 2, log2_cb_size, 2, idx - 1);
                hls_prediction_unit(s, x0 + cb_size / 2, y0 + cb_size / 2, cb_size / 2, cb_size / 2, log2_cb_size, 3, idx - 1);
                break;
            }
        }

        if (!pcm_flag) {
            int rqt_root_cbf = 1;

            if (lc->cu.pred_mode != MODE_INTRA &&
                !(lc->cu.part_mode == PART_2Nx2N && lc->pu.merge_flag)) {
                rqt_root_cbf = ff_hevc_no_residual_syntax_flag_decode(s);
            }
            if (rqt_root_cbf) {
                const static int cbf[2] = { 0 };
                lc->cu.max_trafo_depth = lc->cu.pred_mode == MODE_INTRA ?
                                         s->sps->max_transform_hierarchy_depth_intra + lc->cu.intra_split_flag :
                                         s->sps->max_transform_hierarchy_depth_inter;
                //处理TU四叉树
                ret = hls_transform_tree(s, x0, y0, x0, y0, x0, y0,
                                         log2_cb_size,
                                         log2_cb_size, 0, 0, cbf, cbf);
                if (ret < 0)
                    return ret;
            } else {
                if (!s->sh.disable_deblocking_filter_flag)
                    ff_hevc_deblocking_boundary_strengths(s, x0, y0, log2_cb_size);
            }
        }
    }

    if (s->pps->cu_qp_delta_enabled_flag && lc->tu.is_cu_qp_delta_coded == 0)
        ff_hevc_set_qPy(s, x0, y0, log2_cb_size);

    x = y_cb * min_cb_width + x_cb;
    for (y = 0; y < length; y++) {
        memset(&s->qp_y_tab[x], lc->qp_y, length);
        x += min_cb_width;
    }

    if(((x0 + (1<<log2_cb_size)) & qp_block_mask) == 0 &&
       ((y0 + (1<<log2_cb_size)) & qp_block_mask) == 0) {
        lc->qPy_pred = lc->qp_y;
    }

    set_ct_depth(s, x0, y0, log2_cb_size, lc->ct_depth);

    return 0;
}
```

从源代码可以看出，hls_coding_unit()主要进行了两个方面的处理：

（1）调用hls_prediction_unit()处理PU。

（2）调用hls_transform_tree()处理TU树。

本文分析第一个函数hls_prediction_unit()中相关的代码。

hls_prediction_unit()

hls_prediction_unit()用于处理PU。该函数的定义如下所示。

```
[cpp]
/*
 * 处理PU单元-运动补偿
 *
 * hls_prediction_unit()参数：
 * x0 : PU左上角x坐标
 * y0 : PU左上角y坐标
```

```
 * nPbW : PU宽度
 * nPbH : PU高度
 * log2_cb_size : CB大小取log2()的值
 * partIdx : PU的索引号-分成4个块的时候取0-3，分成两个块的时候取0和1
 *
 * [例]
 *
 * PART_2NxN:
 * +--------+--------+
 * |                 |
 * |                 |
 * |                 |
 * +--------+--------+
 * |                 |
 * |                 |
 * |                 |
 * +--------+--------+
 *
 * 上方PU：
 * hls_prediction_unit(s, x0, y0,               cb_size, cb_size / 2, log2_cb_size, 0, idx);
 * 下方PU：
 * hls_prediction_unit(s, x0, y0 + cb_size / 2, cb_size, cb_size / 2, log2_cb_size, 1, idx);
 *
 */
static void hls_prediction_unit(HEVCContext *s, int x0, int y0,
                                int nPbW, int nPbH,
                                int log2_cb_size, int partIdx, int idx)
{
#define POS(c_idx, x, y)                                                           \
    &s->frame->data[c_idx][((y) >> s->sps->vshift[c_idx]) * s->frame->linesize[c_idx] + \
                           (((x) >> s->sps->hshift[c_idx]) << s->sps->pixel_shift)]
    HEVCLocalContext *lc = s->HEVClc;
    int merge_idx = 0;
    struct MvField current_mv = {{{ 0 }}};

    int min_pu_width = s->sps->min_pu_width;

    MvField *tab_mvf = s->ref->tab_mvf;
    RefPicList  *refPicList = s->ref->refPicList;
    //参考帧
    HEVCFrame *ref0, *ref1;
    //分别指向Y，U，V分量
    uint8_t *dst0 = POS(0, x0, y0);
    uint8_t *dst1 = POS(1, x0, y0);
    uint8_t *dst2 = POS(2, x0, y0);

    int log2_min_cb_size = s->sps->log2_min_cb_size;
    int min_cb_width     = s->sps->min_cb_width;
    int x_cb             = x0 >> log2_min_cb_size;
    int y_cb             = y0 >> log2_min_cb_size;
    int x_pu, y_pu;
    int i, j;

    int skip_flag = SAMPLE_CTB(s->skip_flag, x_cb, y_cb);

    if (!skip_flag)
        lc->pu.merge_flag = ff_hevc_merge_flag_decode(s);

    if (skip_flag || lc->pu.merge_flag) {
        //Merge模式
        if (s->sh.max_num_merge_cand > 1)
            merge_idx = ff_hevc_merge_idx_decode(s);
        else
            merge_idx = 0;

        ff_hevc_luma_mv_merge_mode(s, x0, y0, nPbW, nPbH, log2_cb_size,
                                   partIdx, merge_idx, ¤t_mv);
    } else {
        //AMVP模式
        hevc_luma_mv_mpv_mode(s, x0, y0, nPbW, nPbH, log2_cb_size,
                              partIdx, merge_idx, ¤t_mv);
    }

    x_pu = x0 >> s->sps->log2_min_pu_size;
    y_pu = y0 >> s->sps->log2_min_pu_size;

    for (j = 0; j < nPbH >> s->sps->log2_min_pu_size; j++)
        for (i = 0; i < nPbW >> s->sps->log2_min_pu_size; i++)
            tab_mvf[(y_pu + j) * min_pu_width + x_pu + i] = current_mv;
    //参考了List0
    if (current_mv.pred_flag & PF_L0) {
        ref0 = refPicList[0].ref[current_mv.ref_idx[0]];
        if (!ref0)
            return;
        hevc_await_progress(s, ref0, ¤t_mv.mv[0], y0, nPbH);
    }
    //参考了List1
    if (current_mv.pred_flag & PF_L1) {
        ref1 = refPicList[1].ref[current_mv.ref_idx[1]];
        if (!ref1)
            return;
```

```
        hevc_await_progress(s, ref1, ¤t_mv.mv[1], y0, nPbH);
    }

    if (current_mv.pred_flag == PF_L0) {
        int x0_c = x0 >> s->sps->hshift[1];
        int y0_c = y0 >> s->sps->vshift[1];
        int nPbW_c = nPbW >> s->sps->hshift[1];
        int nPbH_c = nPbH >> s->sps->vshift[1];
        //亮度运动补偿-单向
        luma_mc_uni(s, dst0, s->frame->linesize[0], ref0->frame,
                    ¤t_mv.mv[0], x0, y0, nPbW, nPbH,
                    s->sh.luma_weight_l0[current_mv.ref_idx[0]],
                    s->sh.luma_offset_l0[current_mv.ref_idx[0]]);
        //色度运动补偿
        chroma_mc_uni(s, dst1, s->frame->linesize[1], ref0->frame->data[1], ref0->frame->linesize[1],
                      0, x0_c, y0_c, nPbW_c, nPbH_c, ¤t_mv,
                      s->sh.chroma_weight_l0[current_mv.ref_idx[0]][0], s->sh.chroma_offset_l0[current_mv.ref_idx[0]][0]);
        chroma_mc_uni(s, dst2, s->frame->linesize[2], ref0->frame->data[2], ref0->frame->linesize[2],
                      0, x0_c, y0_c, nPbW_c, nPbH_c, ¤t_mv,
                      s->sh.chroma_weight_l0[current_mv.ref_idx[0]][1], s->sh.chroma_offset_l0[current_mv.ref_idx[0]][1]);
    } else if (current_mv.pred_flag == PF_L1) {
        int x0_c = x0 >> s->sps->hshift[1];
        int y0_c = y0 >> s->sps->vshift[1];
        int nPbW_c = nPbW >> s->sps->hshift[1];
        int nPbH_c = nPbH >> s->sps->vshift[1];

        luma_mc_uni(s, dst0, s->frame->linesize[0], ref1->frame,
                    ¤t_mv.mv[1], x0, y0, nPbW, nPbH,
                    s->sh.luma_weight_l1[current_mv.ref_idx[1]],
                    s->sh.luma_offset_l1[current_mv.ref_idx[1]]);

        chroma_mc_uni(s, dst1, s->frame->linesize[1], ref1->frame->data[1], ref1->frame->linesize[1],
                      1, x0_c, y0_c, nPbW_c, nPbH_c, ¤t_mv,
                      s->sh.chroma_weight_l1[current_mv.ref_idx[1]][0], s->sh.chroma_offset_l1[current_mv.ref_idx[1]][0]);

        chroma_mc_uni(s, dst2, s->frame->linesize[2], ref1->frame->data[2], ref1->frame->linesize[2],
                      1, x0_c, y0_c, nPbW_c, nPbH_c, ¤t_mv,
                      s->sh.chroma_weight_l1[current_mv.ref_idx[1]][1], s->sh.chroma_offset_l1[current_mv.ref_idx[1]][1]);
    } else if (current_mv.pred_flag == PF_BI) {
        int x0_c = x0 >> s->sps->hshift[1];
        int y0_c = y0 >> s->sps->vshift[1];
        int nPbW_c = nPbW >> s->sps->hshift[1];
        int nPbH_c = nPbH >> s->sps->vshift[1];
        //亮度运动补偿-双向
        luma_mc_bi(s, dst0, s->frame->linesize[0], ref0->frame,
                   ¤t_mv.mv[0], x0, y0, nPbW, nPbH,
                   ref1->frame, ¤t_mv.mv[1], ¤t_mv);

        chroma_mc_bi(s, dst1, s->frame->linesize[1], ref0->frame, ref1->frame,
                     x0_c, y0_c, nPbW_c, nPbH_c, ¤t_mv, 0);

        chroma_mc_bi(s, dst2, s->frame->linesize[2], ref0->frame, ref1->frame,
                     x0_c, y0_c, nPbW_c, nPbH_c, ¤t_mv, 1);
    }
}
```

从源代码可以看出，hls_prediction_unit()完成了以下两步工作：

（1）解析码流得到运动矢量。HEVC中包含了Merge和AMVP两种运动矢量预测技术。对于使用Merge的码流，调用ff_hevc_luma_mv_merge_mode()；对于使用AMVP的码流，调用hevc_luma_mv_mpv_mode()。

（2）根据运动矢量进行运动补偿。对于单向预测亮度运动补偿，调用luma_mc_uni()，对于单向预测色度运动补偿，调用chroma_mc_uni()；对于双向预测亮度运动补偿，调用luma_mc_bi()，对于单向预测色度运动补偿，调用chroma_mc_bi()。

luma_mc_uni()

luma_mc_uni()是单向预测亮度运动补偿函数。该函数的定义如下所示。

```cpp
/**
 * 8.5.3.2.2.1 Luma sample unidirectional interpolation process
 *
 * @param s HEVC decoding context
 * @param dst target buffer for block data at block position
 * @param dststride stride of the dst buffer
 * @param ref reference picture buffer at origin (0, 0)
 * @param mv motion vector (relative to block position) to get pixel data from
 * @param x_off horizontal position of block from origin (0, 0)
 * @param y_off vertical position of block from origin (0, 0)
 * @param block_w width of block
 * @param block_h height of block
 * @param luma_weight weighting factor applied to the luma prediction
 * @param luma_offset additive offset applied to the luma prediction value
 */
//亮度运动补偿-单向
static void luma_mc_uni(HEVCContext *s, uint8_t *dst, ptrdiff_t dststride,
                        AVFrame *ref, const Mv *mv, int x_off, int y_off,
                        int block_w, int block_h, int luma_weight, int luma_offset)
{
    HEVCLocalContext *lc = s->HEVClc;
    uint8_t *src         = ref->data[0];
    ptrdiff_t srcstride  = ref->linesize[0];
    int pic_width        = s->sps->width;
    int pic_height       = s->sps->height;
    //亚像素的运动矢量
    //mv0,mv1单位是1/4像素，与00000011相与之后保留后两位
    int mx               = mv->x & 3;
    int my               = mv->y & 3;
    int weight_flag      = (s->sh.slice_type == P_SLICE && s->pps->weighted_pred_flag) ||
                           (s->sh.slice_type == B_SLICE && s->pps->weighted_bipred_flag);
    int idx              = ff_hevc_pel_weight[block_w];

    //整像素的偏移值
    //mv0,mv1单位是1/4像素，在这里除以4之后单位变成整像素
    x_off += mv->x >> 2;
    y_off += mv->y >> 2;
    src   += y_off * srcstride + (x_off << s->sps->pixel_shift);
    //边界处处理
    if (x_off < QPEL_EXTRA_BEFORE || y_off < QPEL_EXTRA_AFTER ||
        x_off >= pic_width - block_w - QPEL_EXTRA_AFTER ||
        y_off >= pic_height - block_h - QPEL_EXTRA_AFTER) {
        const int edge_emu_stride = EDGE_EMU_BUFFER_STRIDE << s->sps->pixel_shift;
        int offset     = QPEL_EXTRA_BEFORE * srcstride       + (QPEL_EXTRA_BEFORE << s->sps->pixel_shift);
        int buf_offset = QPEL_EXTRA_BEFORE * edge_emu_stride + (QPEL_EXTRA_BEFORE << s->sps->pixel_shift);

        s->vdsp.emulated_edge_mc(lc->edge_emu_buffer, src - offset,
                                 edge_emu_stride, srcstride,
                                 block_w + QPEL_EXTRA,
                                 block_h + QPEL_EXTRA,
                                 x_off - QPEL_EXTRA_BEFORE, y_off - QPEL_EXTRA_BEFORE,
                                 pic_width, pic_height);
        src = lc->edge_emu_buffer + buf_offset;
        srcstride = edge_emu_stride;
    }
    //运动补偿
    if (!weight_flag)//普通的
        s->hevcdsp.put_hevc_qpel_uni[idx][!!my][!!mx](dst, dststride, src, srcstride,
                                                      block_h, mx, my, block_w);
    else//加权的
        s->hevcdsp.put_hevc_qpel_uni_w[idx][!!my][!!mx](dst, dststride, src, srcstride,
                                                        block_h, s->sh.luma_log2_weight_denom,
                                                        luma_weight, luma_offset, mx, my, block_w);
}
```

从源代码可以看出，luma_mc_uni()调用了HEVCDSPContext的put_hevc_qpel_uni()汇编函数完成了运动补偿。

luma_mc_bi()

luma_mc_bi()是双向预测亮度运动补偿函数。该函数的定义如下所示。

```cpp
/**
 * 8.5.3.2.2.1 Luma sample bidirectional interpolation process
 *
 * @param s HEVC decoding context
 * @param dst target buffer for block data at block position
 * @param dststride stride of the dst buffer
 * @param ref0 reference picture0 buffer at origin (0, 0)
 * @param mv0 motion vector0 (relative to block position) to get pixel data from
 * @param x_off horizontal position of block from origin (0, 0)
 * @param y_off vertical position of block from origin (0, 0)
 * @param block_w width of block
 * @param block_h height of block
 * @param ref1 reference picture1 buffer at origin (0, 0)
 * @param mv1 motion vector1 (relative to block position) to get pixel data from
```

```
 * @param current_mv current motion vector structure
 */
//亮度运动补偿-双向
 static void luma_mc_bi(HEVCContext *s, uint8_t *dst, ptrdiff_t dststride,
                        AVFrame *ref0, const Mv *mv0, int x_off, int y_off,
                        int block_w, int block_h, AVFrame *ref1, const Mv *mv1, struct MvField *current_mv)
{
    HEVCLocalContext *lc = s->HEVClc;
    ptrdiff_t src0stride  = ref0->linesize[0];
    ptrdiff_t src1stride  = ref1->linesize[0];
    int pic_width        = s->sps->width;
    int pic_height       = s->sps->height;
    //亚像素的运动矢量
    //mv0,mv1单位是1/4像素，与00000011相与之后保留后两位
    int mx0              = mv0->x & 3;
    int my0              = mv0->y & 3;
    int mx1              = mv1->x & 3;
    int my1              = mv1->y & 3;
    int weight_flag      = (s->sh.slice_type == P_SLICE && s->pps->weighted_pred_flag) ||
                           (s->sh.slice_type == B_SLICE && s->pps->weighted_bipred_flag);
    //整像素的偏移值
    //mv0,mv1单位是1/4像素，在这里除以4之后单位变成整像素
    int x_off0           = x_off + (mv0->x >> 2);
    int y_off0           = y_off + (mv0->y >> 2);
    int x_off1           = x_off + (mv1->x >> 2);
    int y_off1           = y_off + (mv1->y >> 2);
    int idx              = ff_hevc_pel_weight[block_w];

    //匹配块数据（整像素精度，没有进行差值）
    //list0
    uint8_t *src0  = ref0->data[0] + y_off0 * src0stride + (int)((unsigned)x_off0 << s->sps->pixel_shift);
    //list1
    uint8_t *src1  = ref1->data[0] + y_off1 * src1stride + (int)((unsigned)x_off1 << s->sps->pixel_shift);
    //边界位置的处理
    if (x_off0 < QPEL_EXTRA_BEFORE || y_off0 < QPEL_EXTRA_AFTER ||
        x_off0 >= pic_width - block_w - QPEL_EXTRA_AFTER ||
        y_off0 >= pic_height - block_h - QPEL_EXTRA_AFTER) {
        const int edge_emu_stride = EDGE_EMU_BUFFER_STRIDE << s->sps->pixel_shift;
        int offset     = QPEL_EXTRA_BEFORE * src0stride       + (QPEL_EXTRA_BEFORE << s->sps->pixel_shift);
        int buf_offset = QPEL_EXTRA_BEFORE * edge_emu_stride + (QPEL_EXTRA_BEFORE << s->sps->pixel_shift);

        s->vdsp.emulated_edge_mc(lc->edge_emu_buffer, src0 - offset,
                                 edge_emu_stride, src0stride,
                                 block_w + QPEL_EXTRA,
                                 block_h + QPEL_EXTRA,
                                 x_off0 - QPEL_EXTRA_BEFORE, y_off0 - QPEL_EXTRA_BEFORE,
                                 pic_width, pic_height);
        src0 = lc->edge_emu_buffer + buf_offset;
        src0stride = edge_emu_stride;
    }

    if (x_off1 < QPEL_EXTRA_BEFORE || y_off1 < QPEL_EXTRA_AFTER ||
        x_off1 >= pic_width - block_w - QPEL_EXTRA_AFTER ||
        y_off1 >= pic_height - block_h - QPEL_EXTRA_AFTER) {
        const int edge_emu_stride = EDGE_EMU_BUFFER_STRIDE << s->sps->pixel_shift;
        int offset     = QPEL_EXTRA_BEFORE * src1stride       + (QPEL_EXTRA_BEFORE << s->sps->pixel_shift);
        int buf_offset = QPEL_EXTRA_BEFORE * edge_emu_stride + (QPEL_EXTRA_BEFORE << s->sps->pixel_shift);

        s->vdsp.emulated_edge_mc(lc->edge_emu_buffer2, src1 - offset,
                                 edge_emu_stride, src1stride,
                                 block_w + QPEL_EXTRA,
                                 block_h + QPEL_EXTRA,
                                 x_off1 - QPEL_EXTRA_BEFORE, y_off1 - QPEL_EXTRA_BEFORE,
                                 pic_width, pic_height);
        src1 = lc->edge_emu_buffer2 + buf_offset;
        src1stride = edge_emu_stride;
    }

    //双向预测分成2步：
    //  (1)使用list0中的匹配块进行单向预测
    //  (2)使用list1中的匹配块再次进行单向预测，然后与第1次预测的结果求平均

    //第1步
    s->hevcdsp.put_hevc_qpel[idx][!!my0][!!mx0](lc->tmp, src0, src0stride,
                                                block_h, mx0, my0, block_w);

    //第2步
    if (!weight_flag)
        s->hevcdsp.put_hevc_qpel_bi[idx][!!my1][!!mx1](dst, dststride, src1, src1stride, lc->tmp,
                                                       block_h, mx1, my1, block_w);
    else
        s->hevcdsp.put_hevc_qpel_bi_w[idx][!!my1][!!mx1](dst, dststride, src1, src1stride, lc->tmp,
                                                         block_h, s->sh.luma_log2_weight_denom,
                                                         s->sh.luma_weight_l0[current_mv->ref_idx[0]],
                                                         s->sh.luma_weight_l1[current_mv->ref_idx[1]],
                                                         s->sh.luma_offset_l0[current_mv->ref_idx[0]],
                                                         s->sh.luma_offset_l1[current_mv->ref_idx[1]],
                                                         mx1, my1, block_w);

}
```

从源代码可以看出，luma_mc_bi()调用了HEVCDSPContext的put_hevc_qpel()和put_hevc_qpel_bi()汇编函数完成了运动补偿。后文将会对这些运动补偿汇编函数进行分析。

运动补偿相关知识

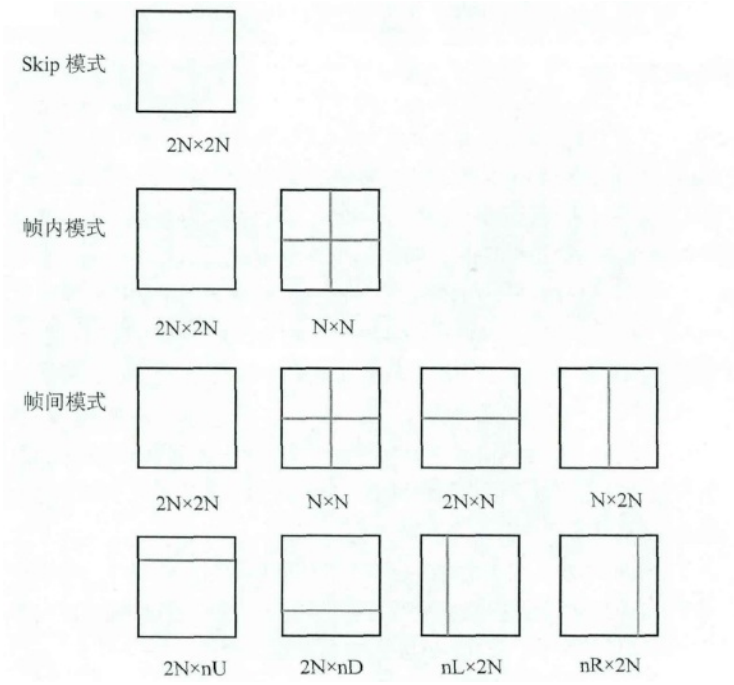
本部分简单总结运动补偿相关的知识，并举几个例子。

运动补偿小知识

本节简单回顾一下《HEVC标准》中有关运动补偿的知识。

PU的划分

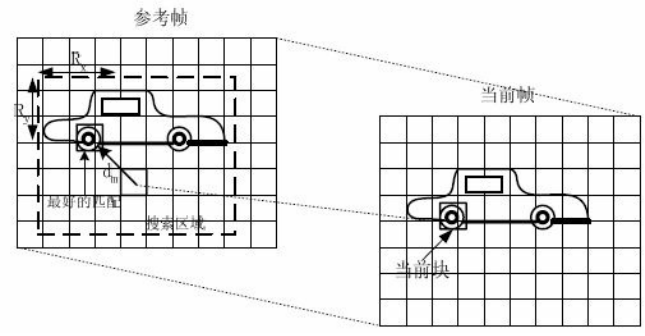
HEVC中CU支持如下几种划分方式。可以看出帧内CU只有2种划分模式，而帧间CU支持8种划分模式（其中后4种是非对称划分模式）。



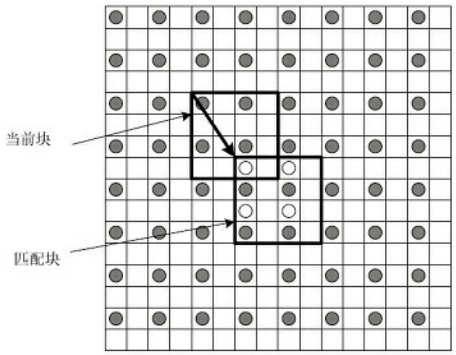
1/4像素运动估计

运动估计的理论基础就是活动图像邻近帧中的景物存在着一定的相关性。因此在压缩编码中不需要传递每一帧的所有信息，而只需要传递帧与帧之间差值就可以了（可以想象，如果画面背景是静止的，那么只需要传递很少的数据）。

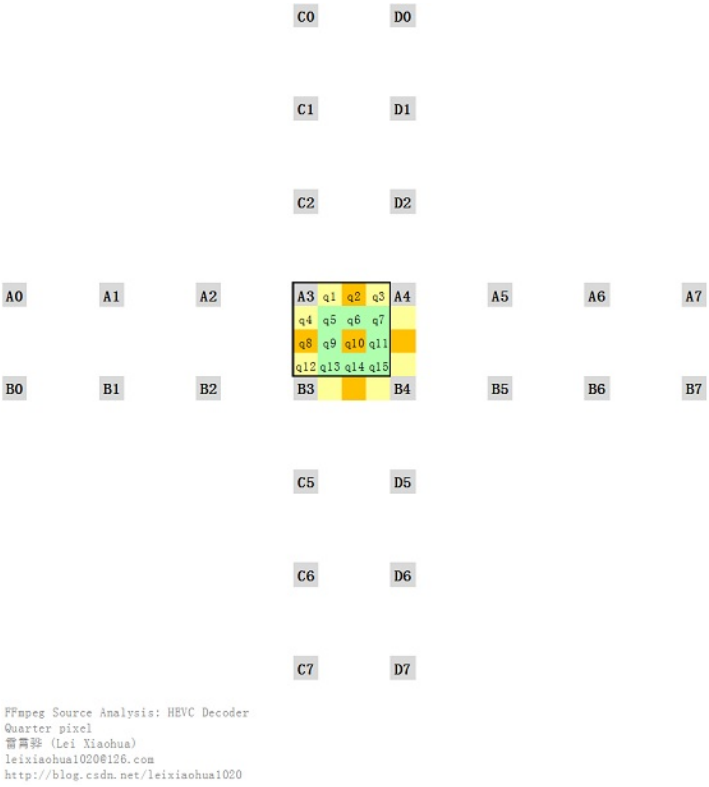
在视频编码的运动估计步骤中，会查找与当前宏块或者子宏块“长得像”的宏块作为“匹配块”，然后编码传输匹配块的位置（运动矢量，参考帧）和当前宏块与匹配块之间的微小差别（残差数据）。例如下图中，当前宏块中一个“车轮”在参考帧中找到了形状同样为一个“轮子”的匹配块。



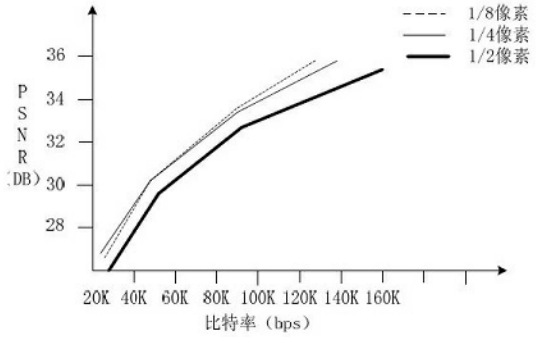
最早视频编码标准中都是以整像素的方式进行运动估计的。这样处理的好处是计算简单，坏处是不够精确。随着硬件技术的进步，比较新的视频编码标准（例如MPEG2）中使用1/2像素精度的方式进行运动估计。这样做计算相对复杂，但是计算也相对准确。1/2像素精度运动估计如下图所示。



H.264中对运动估计的精度要求又有了提升，变成了1/4像素精度。HEVC在运动估计方面同样使用了1/4精度。在H.264 / HEVC编码和解码的过程中，需要将画面中的像素进行插值——简单地说就是把原先的1个像素点拓展成4x4一共16个点。下图显示了HEVC编码和解码过程中像素插值情况。可以看出原先的A3点的右下方通过插值的方式产生了q1--q15一共15个点。



一些实验证明，1/4像素精度基本上达到了运动估计性能提升的极限。更高精度的运动估计并不能更明显的提升性能，却会导致计算复杂度的显著提升。因此现存主流的编解码标准在运动估计方面都采用了1/4精度。曾经有人压缩对比过1/2、1/4、1/8精度的运动估计下编码的视频质量，如下图所示。



从图中可以看出：1/4精度相比于1/2精度来说有显著的提升，但是1/8精度实际上和1/4精度是差不多的。

四分之一像素内插方式

HEVC的1/4像素内插的方法和H.264是不一样的。H.264首先通过6抽头的滤波器获得半像素点，然后通过线性内插的方式获得1/4像素点。HEVC则在半像素点使用了8抽头的滤波器，在1/4像素点使用了7抽头的滤波器。以上面四分之一像素插值示意图为例，分别记录一下H.264和HEVC各个差值点的计算方法。

【H.264像素插值方式】

H.264的水平半像素点q2插值公式为：

$$q2= round((A1 – 5*A2 + 20*A3 + 20*A4 – 5*A5 + A6) / 32)$$

PS：垂直半像素点q8，对角线半像素点q10的计算方法是类似的。

H.264的1/4像素点q1插值公式为：

$$q1=round((A3+q2)/2)$$

PS：其它1/4像素点的计算方法也是类似的，都是通过整像素点和半像素点线性内插获得。

【HEVC像素插值方式】

HEVC的半像素点q2插值公式为：

$$q2= round((–A0 + 4*A1 – 10*A2 + 58*A3 + 17*A4 – 5*A5 + A6) / 32)$$

HEVC的1/4像素点q1插值公式为：

$$q1= round((–A0 + 4*A1 – 11*A2 + 40*A3+ 40*A4 – 11*A5 + 4*A6 – A7) / 32)$$

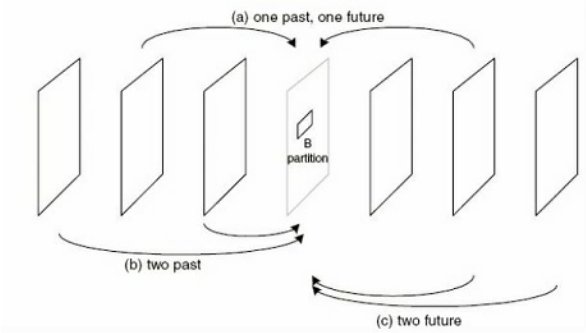
HEVC的3/4像素点q3插值公式为：

$$q3= round((A1 – 5*A2 + 17*A3 + 58*A4 – 10*A5 + 4*A6 – A7) / 32)$$

PS：其它1/4像素点的计算方法也是类似的。

单向预测与双向预测

在运动估计的过程中，不仅仅只可以选择一个图像作为参考帧（P帧），而且还可以选择两张图片作为参考帧（B帧）。使用一张图像作为参考帧称为单向预测，而使用一张图像作为参考帧称为双向预测。使用单向预测的时候，直接将参考帧上的匹配块的数据“搬移下来”作后续的处理（“赋值”），而使用双向预测的时候，需要首先将两个参考帧上的匹配块的数据求平均值（“求平均”），然后再做后续处理。毫无疑问双向预测可以得到更好的压缩效果，但是也会使码流变得复杂一些。双向预测的示意图如下所示。

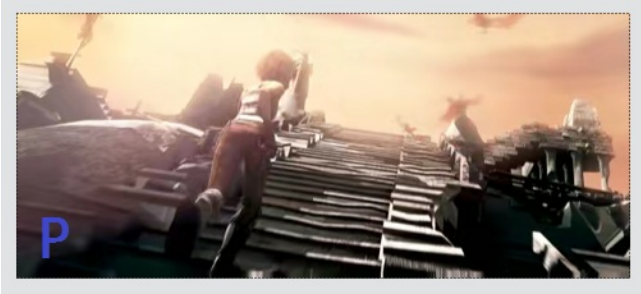


帧间预测实例

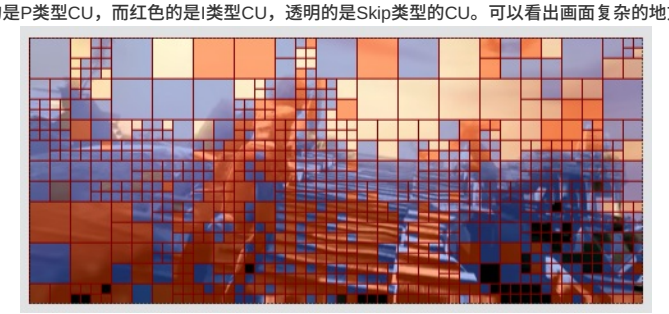
本节以一段《Sintel》动画的码流为例，看一下HEVC码流中的运动补偿相关的信息。

【示例1-P帧】

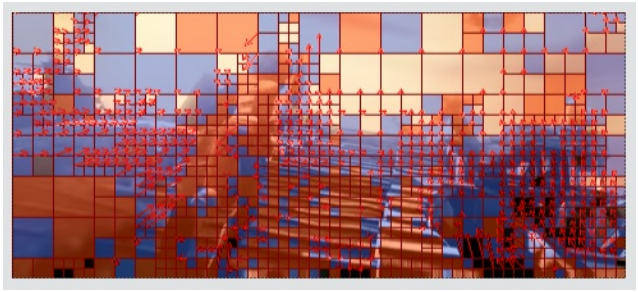
下图为一个P帧解码后的图像。



下图为该帧CTU的划分方式。其中蓝色的是P类型CU，而红色的是I类型CU，透明的是Skip类型的CU。可以看出画面复杂的地方CTU划分比较细。

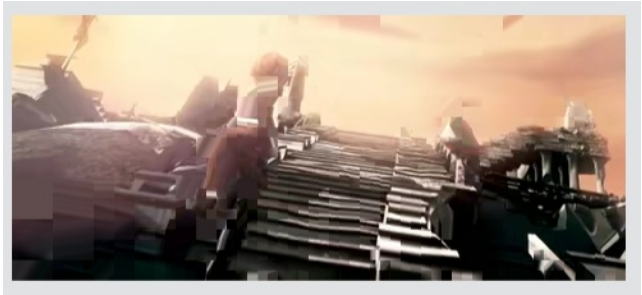


下图为每个P类型的CU的运动矢量信息。

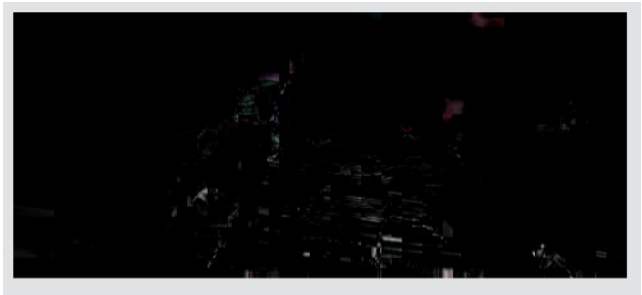


下图显示了运动矢量与图像内容变化之间的关系。

下图为经过运动补偿，没有进行残差叠加处理的视频内容。

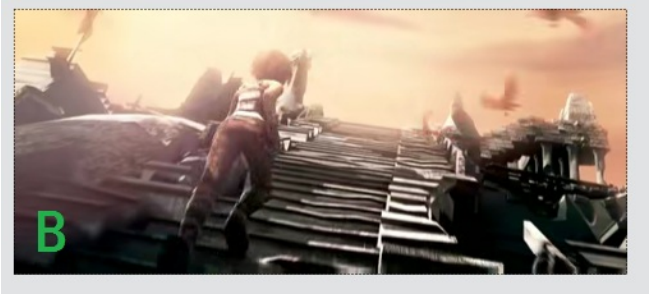


下图为该帧的残差信息。

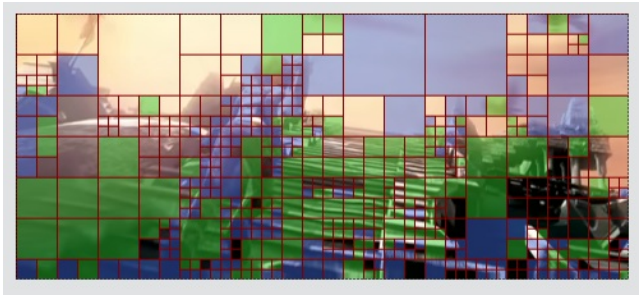


【示例2-B帧】

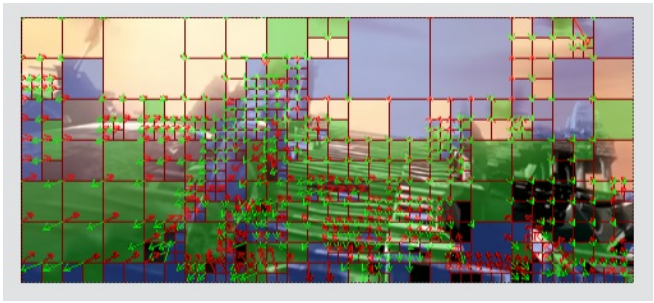
下图为一个B帧解码后的图像。



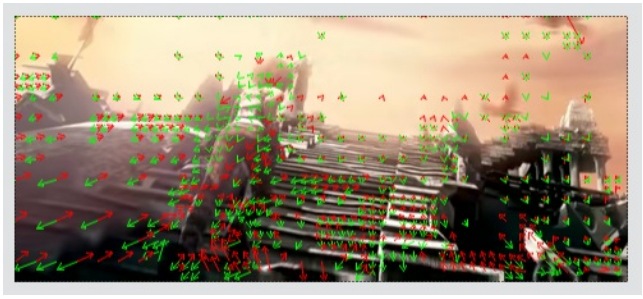
下图为该帧CTU的划分方式。其中蓝色的是P类型CU，而绿色的是B类型CU，透明的是Skip类型的CU。可以看出画面复杂的地方CTU划分比较细。



下图为每个CU的运动矢量信息。

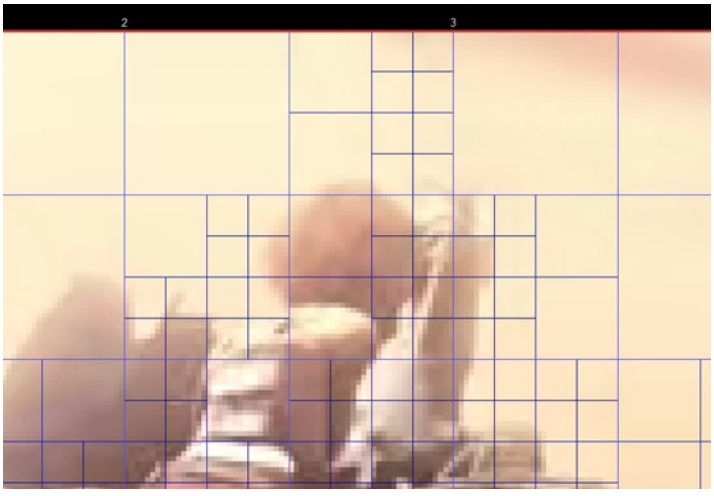


下图显示了运动矢量与图像内容变化之间的关系。

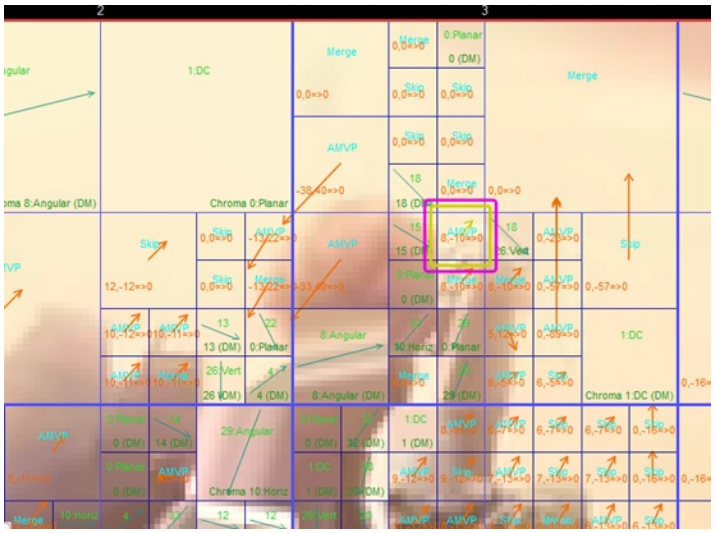


【示例3】

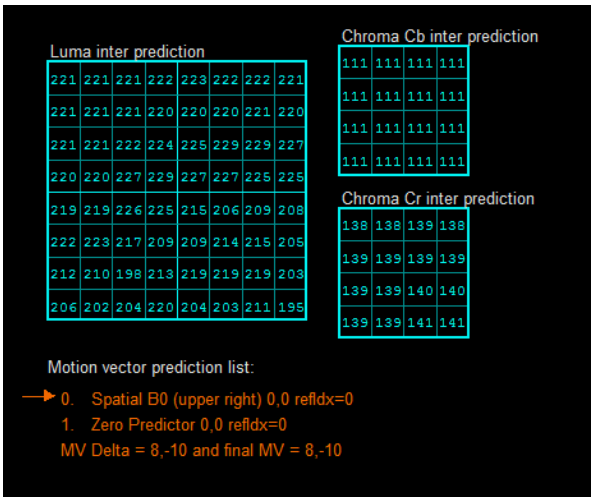
本节以一段《Sintel》动画的码流为例，看一下HEVC码流中帧间预测具体的信息。下图为一个P帧解码后的图像。



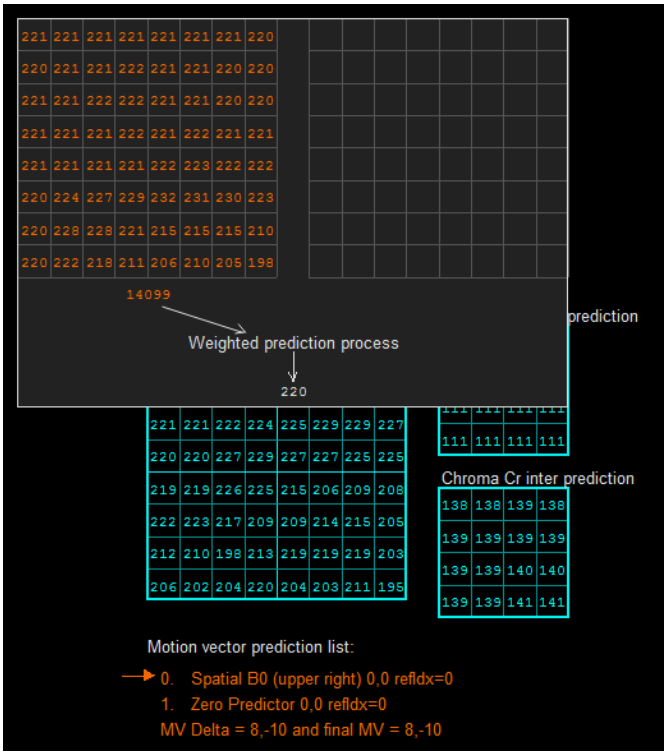
下图显示了该帧图像的帧间预测模式（部分CU采用了帧内预测模式，在这里不分析）。在这里我们选择一个8x8 CU（图中以紫色方框标出）看一下其中具体的信息。该CU采用了AMVP帧间预测方式，运动矢量为(8,-10)。



该8x8CU的像素值如下图所示。



下图显示了该CU中一个像素的计算方式。从图中可以看出，每个像素值通过匹配块周围的8x8=64个像素内插获得结果。图中的像素内插后的结果是“220”。



运动补偿汇编函数源代码

运动补偿相关的汇编函数位于HEVCDSPContext中。HEVCDSPContext的初始化函数是ff_hevc_dsp_init()。该函数对HEVCDSPContext结构体中的函数指针进行了赋值。FFmpeg HEVC解码器运行的过程中只要调用HEVCDSPContext的函数指针就可以完成相应的功能。

ff_hevc_dsp_init()

ff_hevc_dsp_init()用于初始化HEVCDSPContext结构体中的汇编函数指针。该函数的定义如下所示。

```cpp
void ff_hevc_dsp_init(HEVCDSPContext *hevcdsp, int bit_depth)
{
#undef FUNC
#define FUNC(a, depth) a ## _ ## depth

#undef PEL_FUNC
#define PEL_FUNC(dst1, idx1, idx2, a, depth)                                    \
    for(i = 0 ; i < 10 ; i++)                                                   \
{                                                                               \
    hevcdsp->dst1[i][idx1][idx2] = a ## _ ## depth;                             \
}

#undef EPEL_FUNCS
#define EPEL_FUNCS(depth)                                                   \
    PEL_FUNC(put_hevc_epel, 0, 0, put_hevc_pel_pixels, depth);              \
    PEL_FUNC(put_hevc_epel, 0, 1, put_hevc_epel_h, depth);                  \
    PEL_FUNC(put_hevc_epel, 1, 0, put_hevc_epel_v, depth);                  \
    PEL_FUNC(put_hevc_epel, 1, 1, put_hevc_epel_hv, depth)

#undef EPEL_UNI_FUNCS
#define EPEL_UNI_FUNCS(depth)                                               \
    PEL_FUNC(put_hevc_epel_uni, 0, 0, put_hevc_pel_uni_pixels, depth);      \
    PEL_FUNC(put_hevc_epel_uni, 0, 1, put_hevc_epel_uni_h, depth);          \
    PEL_FUNC(put_hevc_epel_uni, 1, 0, put_hevc_epel_uni_v, depth);          \
    PEL_FUNC(put_hevc_epel_uni, 1, 1, put_hevc_epel_uni_hv, depth);         \
    PEL_FUNC(put_hevc_epel_uni_w, 0, 0, put_hevc_pel_uni_w_pixels, depth);  \
    PEL_FUNC(put_hevc_epel_uni_w, 0, 1, put_hevc_epel_uni_w_h, depth);      \
    PEL_FUNC(put_hevc_epel_uni_w, 1, 0, put_hevc_epel_uni_w_v, depth);      \
    PEL_FUNC(put_hevc_epel_uni_w, 1, 1, put_hevc_epel_uni_w_hv, depth)

#undef EPEL_BI_FUNCS
#define EPEL_BI_FUNCS(depth)                                                \
    PEL_FUNC(put_hevc_epel_bi, 0, 0, put_hevc_pel_bi_pixels, depth);        \
    PEL_FUNC(put_hevc_epel_bi, 0, 1, put_hevc_epel_bi_h, depth);            \
    PEL_FUNC(put_hevc_epel_bi, 1, 0, put_hevc_epel_bi_v, depth);            \
    PEL_FUNC(put_hevc_epel_bi, 1, 1, put_hevc_epel_bi_hv, depth);           \
    PEL_FUNC(put_hevc_epel_bi_w, 0, 0, put_hevc_pel_bi_w_pixels, depth);    \
    PEL_FUNC(put_hevc_epel_bi_w, 0, 1, put_hevc_epel_bi_w_h, depth);        \
    PEL_FUNC(put_hevc_epel_bi_w, 1, 0, put_hevc_epel_bi_w_v, depth);        \
    PEL_FUNC(put_hevc_epel_bi_w, 1, 1, put_hevc_epel_bi_w_hv, depth)

#undef QPEL_FUNCS
#define QPEL_FUNCS(depth)                                                   \
    PEL_FUNC(put_hevc_qpel, 0, 0, put_hevc_pel_pixels, depth);              \
    PEL_FUNC(put_hevc_qpel, 0, 1, put_hevc_qpel_h, depth);                  \
    PEL_FUNC(put_hevc_qpel, 1, 0, put_hevc_qpel_v, depth);                  \
    PEL_FUNC(put_hevc_qpel, 1, 1, put_hevc_qpel_hv, depth)

#undef QPEL_UNI_FUNCS
#define QPEL_UNI_FUNCS(depth)                                               \
    PEL_FUNC(put_hevc_qpel_uni, 0, 0, put_hevc_pel_uni_pixels, depth);      \
    PEL_FUNC(put_hevc_qpel_uni, 0, 1, put_hevc_qpel_uni_h, depth);          \
    PEL_FUNC(put_hevc_qpel_uni, 1, 0, put_hevc_qpel_uni_v, depth);          \
    PEL_FUNC(put_hevc_qpel_uni, 1, 1, put_hevc_qpel_uni_hv, depth);         \
    PEL_FUNC(put_hevc_qpel_uni_w, 0, 0, put_hevc_pel_uni_w_pixels, depth);  \
    PEL_FUNC(put_hevc_qpel_uni_w, 0, 1, put_hevc_qpel_uni_w_h, depth);      \
    PEL_FUNC(put_hevc_qpel_uni_w, 1, 0, put_hevc_qpel_uni_w_v, depth);      \
    PEL_FUNC(put_hevc_qpel_uni_w, 1, 1, put_hevc_qpel_uni_w_hv, depth)

#undef QPEL_BI_FUNCS
#define QPEL_BI_FUNCS(depth)                                                \
    PEL_FUNC(put_hevc_qpel_bi, 0, 0, put_hevc_pel_bi_pixels, depth);        \
    PEL_FUNC(put_hevc_qpel_bi, 0, 1, put_hevc_qpel_bi_h, depth);            \
    PEL_FUNC(put_hevc_qpel_bi, 1, 0, put_hevc_qpel_bi_v, depth);            \
    PEL_FUNC(put_hevc_qpel_bi, 1, 1, put_hevc_qpel_bi_hv, depth);           \
    PEL_FUNC(put_hevc_qpel_bi_w, 0, 0, put_hevc_pel_bi_w_pixels, depth);    \
    PEL_FUNC(put_hevc_qpel_bi_w, 0, 1, put_hevc_qpel_bi_w_h, depth);        \
    PEL_FUNC(put_hevc_qpel_bi_w, 1, 0, put_hevc_qpel_bi_w_v, depth);        \
    PEL_FUNC(put_hevc_qpel_bi_w, 1, 1, put_hevc_qpel_bi_w_hv, depth)

#define HEVC_DSP(depth)                                                 \
    hevcdsp->put_pcm                = FUNC(put_pcm, depth);             \
    hevcdsp->transform_add[0]       = FUNC(transform_add4x4, depth);    \
    hevcdsp->transform_add[1]       = FUNC(transform_add8x8, depth);    \
    hevcdsp->transform_add[2]       = FUNC(transform_add16x16, depth);  \
    hevcdsp->transform_add[3]       = FUNC(transform_add32x32, depth);  \
    hevcdsp->transform_skip         = FUNC(transform_skip, depth);      \
    hevcdsp->transform_rdpcm        = FUNC(transform_rdpcm, depth);     \
    hevcdsp->idct_4x4_luma          = FUNC(transform_4x4_luma, depth);  \
    hevcdsp->idct[0]                = FUNC(idct_4x4, depth);            \
```

```
    hevcdsp->idct[1]                = FUNC(idct_8x8, depth);                \
    hevcdsp->idct[2]                = FUNC(idct_16x16, depth);              \
    hevcdsp->idct[3]                = FUNC(idct_32x32, depth);              \
                                                                            \
    hevcdsp->idct_dc[0]             = FUNC(idct_4x4_dc, depth);             \
    hevcdsp->idct_dc[1]             = FUNC(idct_8x8_dc, depth);             \
    hevcdsp->idct_dc[2]             = FUNC(idct_16x16_dc, depth);           \
    hevcdsp->idct_dc[3]             = FUNC(idct_32x32_dc, depth);           \
                                                                            \
    hevcdsp->sao_band_filter    = FUNC(sao_band_filter_0, depth);              \
    hevcdsp->sao_edge_filter[0] = FUNC(sao_edge_filter_0, depth);              \
    hevcdsp->sao_edge_filter[1] = FUNC(sao_edge_filter_1, depth);              \
                                                                               \
    QPEL_FUNCS(depth);                                                         \
    QPEL_UNI_FUNCS(depth);                                                     \
    QPEL_BI_FUNCS(depth);                                                      \
    EPEL_FUNCS(depth);                                                         \
    EPEL_UNI_FUNCS(depth);                                                     \
    EPEL_BI_FUNCS(depth);                                                      \
                                                                               \
    hevcdsp->hevc_h_loop_filter_luma     = FUNC(hevc_h_loop_filter_luma, depth);   \
    hevcdsp->hevc_v_loop_filter_luma     = FUNC(hevc_v_loop_filter_luma, depth);   \
    hevcdsp->hevc_h_loop_filter_chroma   = FUNC(hevc_h_loop_filter_chroma, depth); \
    hevcdsp->hevc_v_loop_filter_chroma   = FUNC(hevc_v_loop_filter_chroma, depth); \
    hevcdsp->hevc_h_loop_filter_luma_c   = FUNC(hevc_h_loop_filter_luma, depth);   \
    hevcdsp->hevc_v_loop_filter_luma_c   = FUNC(hevc_v_loop_filter_luma, depth);   \
    hevcdsp->hevc_h_loop_filter_chroma_c = FUNC(hevc_h_loop_filter_chroma, depth); \
    hevcdsp->hevc_v_loop_filter_chroma_c = FUNC(hevc_v_loop_filter_chroma, depth)
int i = 0;

    switch (bit_depth) {
    case 9:
        HEVC_DSP(9);
        break;
    case 10:
        HEVC_DSP(10);
        break;
    case 12:
        HEVC_DSP(12);
        break;
    default:
        HEVC_DSP(8);
        break;
    }

    if (ARCH_X86)
        ff_hevc_dsp_init_x86(hevcdsp, bit_depth);
}
```

从源代码可以看出，ff_hevc_dsp_init()函数中包含一个名为“HEVC_DSP(depth)”的很长的宏定义。该宏定义中包含了C语言版本的帧内预测函数的初始化代码。ff_hevc_dsp_init()会根据系统的颜色位深bit_depth初始化相应的C语言版本的帧内预测函数。在函数的末尾则包含了汇编函数的初始化函数：如果系统是X86架构的，则会调用ff_hevc_dsp_init_x86()初始化X86平台下经过汇编优化的函数。下面以8bit颜色位深为例，看一下“HEVC_DSP(8)”的展开结果。

```cpp
hevcdsp->put_pcm                = put_pcm_8;
hevcdsp->transform_add[0]       = transform_add4x4_8;
hevcdsp->transform_add[1]       = transform_add8x8_8;
hevcdsp->transform_add[2]       = transform_add16x16_8;
hevcdsp->transform_add[3]       = transform_add32x32_8;
hevcdsp->transform_skip         = transform_skip_8;
hevcdsp->transform_rdpcm        = transform_rdpcm_8;
hevcdsp->idct_4x4_luma          = transform_4x4_luma_8;
hevcdsp->idct[0]                = idct_4x4_8;
hevcdsp->idct[1]                = idct_8x8_8;
hevcdsp->idct[2]                = idct_16x16_8;
hevcdsp->idct[3]                = idct_32x32_8;

hevcdsp->idct_dc[0]             = idct_4x4_dc_8;
hevcdsp->idct_dc[1]             = idct_8x8_dc_8;
hevcdsp->idct_dc[2]             = idct_16x16_dc_8;
hevcdsp->idct_dc[3]             = idct_32x32_dc_8;

hevcdsp->sao_band_filter    = sao_band_filter_0_8;
hevcdsp->sao_edge_filter[0] = sao_edge_filter_0_8;
hevcdsp->sao_edge_filter[1] = sao_edge_filter_1_8;

    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel[i][0][0] = put_hevc_pel_pixels_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel[i][0][1] = put_hevc_qpel_h_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel[i][1][0] = put_hevc_qpel_v_8;
```

```
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel[i][1][1] = put_hevc_qpel_hv_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_uni[i][0][0] = put_hevc_pel_uni_pixels_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_uni[i][0][1] = put_hevc_qpel_uni_h_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_uni[i][1][0] = put_hevc_qpel_uni_v_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_uni[i][1][1] = put_hevc_qpel_uni_hv_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_uni_w[i][0][0] = put_hevc_pel_uni_w_pixels_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_uni_w[i][0][1] = put_hevc_qpel_uni_w_h_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_uni_w[i][1][0] = put_hevc_qpel_uni_w_v_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_uni_w[i][1][1] = put_hevc_qpel_uni_w_hv_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_bi[i][0][0] = put_hevc_pel_bi_pixels_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_bi[i][0][1] = put_hevc_qpel_bi_h_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_bi[i][1][0] = put_hevc_qpel_bi_v_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_bi[i][1][1] = put_hevc_qpel_bi_hv_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_bi_w[i][0][0] = put_hevc_pel_bi_w_pixels_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_bi_w[i][0][1] = put_hevc_qpel_bi_w_h_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_bi_w[i][1][0] = put_hevc_qpel_bi_w_v_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_qpel_bi_w[i][1][1] = put_hevc_qpel_bi_w_hv_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel[i][0][0] = put_hevc_pel_pixels_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel[i][0][1] = put_hevc_epel_h_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel[i][1][0] = put_hevc_epel_v_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel[i][1][1] = put_hevc_epel_hv_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_uni[i][0][0] = put_hevc_pel_uni_pixels_8;
};
    for(i = 0 ; i < 10 ; i++)
{
```

```
    hevcdsp->put_hevc_epel_uni[i][0][1] = put_hevc_epel_uni_h_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_uni[i][1][0] = put_hevc_epel_uni_v_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_uni[i][1][1] = put_hevc_epel_uni_hv_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_uni_w[i][0][0] = put_hevc_pel_uni_w_pixels_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_uni_w[i][0][1] = put_hevc_epel_uni_w_h_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_uni_w[i][1][0] = put_hevc_epel_uni_w_v_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_uni_w[i][1][1] = put_hevc_epel_uni_w_hv_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_bi[i][0][0] = put_hevc_pel_bi_pixels_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_bi[i][0][1] = put_hevc_epel_bi_h_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_bi[i][1][0] = put_hevc_epel_bi_v_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_bi[i][1][1] = put_hevc_epel_bi_hv_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_bi_w[i][0][0] = put_hevc_pel_bi_w_pixels_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_bi_w[i][0][1] = put_hevc_epel_bi_w_h_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_bi_w[i][1][0] = put_hevc_epel_bi_w_v_8;
};
    for(i = 0 ; i < 10 ; i++)
{
    hevcdsp->put_hevc_epel_bi_w[i][1][1] = put_hevc_epel_bi_w_hv_8;
};

    hevcdsp->hevc_h_loop_filter_luma     = hevc_h_loop_filter_luma_8;
    hevcdsp->hevc_v_loop_filter_luma     = hevc_v_loop_filter_luma_8;
    hevcdsp->hevc_h_loop_filter_chroma   = hevc_h_loop_filter_chroma_8;
    hevcdsp->hevc_v_loop_filter_chroma   = hevc_v_loop_filter_chroma_8;
    hevcdsp->hevc_h_loop_filter_luma_c   = hevc_h_loop_filter_luma_8;
    hevcdsp->hevc_v_loop_filter_luma_c   = hevc_v_loop_filter_luma_8;
    hevcdsp->hevc_h_loop_filter_chroma_c = hevc_h_loop_filter_chroma_8;
hevcdsp->hevc_v_loop_filter_chroma_c = hevc_v_loop_filter_chroma_8
```

可以看出“HEVC_DSP(8)”中包含了DCT、IDCT、1/4像素运动补偿、SAO滤波器、去块效应滤波器等模块的C语言版本函数。本文关注其中的1/4像素运动补偿函数。通过上述代码可以总结出下面几个用于像素插值的函数：

HEVCDSPContext->put_hevc_qpel_uni[i][0][1]()：单向预测水平像素插值函数。C语言版本函数为put_hevc_qpel_uni_h_8()

HEVCDSPContext->put_hevc_qpel_uni[i][1][0]()：单向预测垂直像素插值函数。C语言版本函数为put_hevc_qpel_uni_v_8()

HEVCDSPContext->put_hevc_qpel_uni[i][1][1]()：单向预测中心像素插值函数。C语言版本函数为put_hevc_qpel_uni_hv_8()

HEVCDSPContext->put_hevc_qpel_bi[i][0][1]()：双向预测水平像素插值函数。C语言版本函数为put_hevc_qpel_bi_h_8()

HEVCDSPContext->put_hevc_qpel_bi[i][1][0]()：双向预测垂直像素插值函数。C语言版本函数为put_hevc_qpel_bi_v_8()

HEVCDSPContext->put_hevc_qpel_bi[i][1][1]()：双向预测中心像素插值函数。C语言版本函数为put_hevc_qpel_bi_hv_8()

下文例举其中的几个函数进行分析。

put_hevc_qpel_uni_h_8()

put_hevc_qpel_uni_h_8()是单向预测水平像素插值函数。该函数的定义如下所示。

```cpp
/*
 * 单向预测
 * 水平（Horizontal）滤波像素插值
 *
 *
 *  A   B   C   D X E   F   G   H
 *
 *
 * 参数：
 * _dst：输出的插值后像素
 * _dststride：输出一行像素数据的大小
 * _src：输入的整像素
 * _srcstride：输入一行像素数据的大小
 * height：像素的宽
 * width：像素的高
 * mx：运动矢量亚像素x方向取值。以1/4像素为基本单位
 * my：运动矢量亚像素x方向取值。以1/4像素为基本单位
 *
 */
static void FUNC(put_hevc_qpel_uni_h)(uint8_t *_dst,  ptrdiff_t _dststride,
                                      uint8_t *_src, ptrdiff_t _srcstride,
                                      int height, intptr_t mx, intptr_t my, int width)
{
    int x, y;
    pixel        *src       = (pixel*)_src;
    ptrdiff_t     srcstride = _srcstride / sizeof(pixel);
    pixel *dst          = (pixel *)_dst;
    ptrdiff_t dststride = _dststride / sizeof(pixel);
    //ff_hevc_qpel_filters[]是滤波器插值系数数组
    //[0]为1/4像素点插值；[1]为半像素点插值；[2]为3/4像素点插值
    const int8_t *filter    = ff_hevc_qpel_filters[mx - 1];
    int shift = 14 - BIT_DEPTH;

#if BIT_DEPTH < 14
    int offset = 1 << (shift - 1);
#else
    int offset = 0;
#endif

    //处理x*y个像素
    //注意，调用了QPEL_FILTER(),其中使用filter[]中的系数进行滤波。
    //QPEL_FILTER()的参数是(src, 1)
    //其中第2个参数stride代表用于滤波的点之前的间距。取1的话是水平滤波，取srcstride的话是垂直滤波
    for (y = 0; y < height; y++) {
        for (x = 0; x < width; x++)
            dst[x] = av_clip_pixel(((QPEL_FILTER(src, 1) >> (BIT_DEPTH - 8)) + offset) >> shift);
        src += srcstride;
        dst += dststride;
    }
}
```

put_hevc_qpel_uni_h_8()源代码中的filter[]用于从静态数组ff_hevc_qpel_filters[3][]中选择一组滤波参数。该数组中一共有3组参数可以选择，分别对应着1/4像素插值点、半像素插值点、3/4像素插值点。ff_hevc_qpel_filters[3][]定义如下所示。

```cpp
//滤波器插值系数数组
//[0]为1/4像素点插值；[1]为半像素点插值；[2]为3/4像素点插值
DECLARE_ALIGNED(16, const int8_t, ff_hevc_qpel_filters[3][16]) = {
    //1/4像素位置
    { -1,  4,-10, 58, 17, -5,  1,  0, -1,  4,-10, 58, 17, -5,  1,  0},
    //半像素位置
    { -1,  4,-11, 40, 40,-11,  4, -1, -1,  4,-11, 40, 40,-11,  4, -1},
    //3/4像素位置
    {  0,  1, -5, 17, 58,-10,  4, -1,  0,  1, -5, 17, 58,-10,  4, -1}
};
```

在选定了滤波参数后，put_hevc_qpel_uni_h_8()就开始逐点对wxh的像素块进行插值。每个点在插值的时候会调用一个宏“QPEL_FILTER(src, 1)”用于进行具体的滤波工作。“QPEL_FILTER(src, stride)”是一个用于滤波的宏，定义如下所示。

```cpp
//半像素插值滤波器
//8个点
//filter[]中存储了系数
#define QPEL_FILTER(src, stride)                                \
    (filter[0] * src[x - 3 * stride] +                          \
     filter[1] * src[x - 2 * stride] +                          \
     filter[2] * src[x -     stride] +                          \
     filter[3] * src[x             ] +                          \
     filter[4] * src[x +     stride] +                          \
     filter[5] * src[x + 2 * stride] +                          \
     filter[6] * src[x + 3 * stride] +                          \
     filter[7] * src[x + 4 * stride])
```

“QPEL_FILTER(src, 1)”展开后的结果如下图所示。

```cpp
av_clip_uint8_c((((filter[0] * src[x - 3 * 1] +
 filter[1] * src[x - 2 * 1] +
 filter[2] * src[x -     1] +
 filter[3] * src[x             ] +
 filter[4] * src[x +     1] +
 filter[5] * src[x + 2 * 1] +
 filter[6] * src[x + 3 * 1] +
 filter[7] * src[x + 4 * 1]) >> (8 - 8)) + offset) >> shift)
```

可以看出QPEL_FILTER()在滤波的点左右共取了8个点进行滤波处理。

put_hevc_qpel_uni_v_8()

put_hevc_qpel_uni_v_8()是单向预测垂直像素插值函数。该函数的定义如下所示。

```cpp
/*
 * 单向预测
 * 垂直（Vertical）滤波像素插值
 *
 *                A
 *
 *                B
 *
 *                C
 *
 *                D
 *                X
 *                E
 *
 *                F
 *
 *                G
 *
 *                H
 *
 * 参数：
 * _dst：输出的插值后像素
 * _dststride：输出一行像素数据的大小
 * _src：输入的整像素
 * _srcstride：输入一行像素数据的大小
 * height：像素的宽
 * width：像素的高
 * mx：运动矢量亚像素x方向取值。以1/4像素为基本单位
 * my：运动矢量亚像素x方向取值。以1/4像素为基本单位
 *
 */
static void FUNC(put_hevc_qpel_uni_v)(uint8_t *_dst,  ptrdiff_t _dststride,
                                      uint8_t *_src, ptrdiff_t _srcstride,
                                      int height, intptr_t mx, intptr_t my, int width)
{
    int x, y;
    pixel        *src       = (pixel*)_src;
    ptrdiff_t     srcstride = _srcstride / sizeof(pixel);
    pixel *dst          = (pixel *)_dst;
    ptrdiff_t dststride = _dststride / sizeof(pixel);
    //ff_hevc_qpel_filters[]是滤波器插值系数数组
    //[0]为1/4像素点插值；[1]为半像素点插值；[2]为3/4像素点插值
    const int8_t *filter    = ff_hevc_qpel_filters[my - 1];
    int shift = 14 - BIT_DEPTH;

#if BIT_DEPTH < 14
    int offset = 1 << (shift - 1);
#else
    int offset = 0;
#endif

    //处理x*y个像素
    //注意，调用了QPEL_FILTER(),其中使用filter[]中的系数进行滤波。
    //QPEL_FILTER()的参数是(src, srcstride)
    //其中第2个参数stride代表用于滤波的点之前的间距。取1的话是水平滤波，取srcstride的话是垂直滤波
    for (y = 0; y < height; y++) {
        for (x = 0; x < width; x++)
            dst[x] = av_clip_pixel(((QPEL_FILTER(src, srcstride) >> (BIT_DEPTH - 8)) + offset) >> shift);
        src += srcstride;
        dst += dststride;
    }
}
```

从源代码可以看出，put_hevc_qpel_uni_v_8()的流程和put_hevc_qpel_uni_h_8()基本上是一模一样的。同样也是先选择一组系数存于filter[]中，然后调用“QPEL_FILTER()”进行滤波。它们之间的区别在于put_hevc_qpel_uni_v_8()中滤波的宏是“QPEL_FILTER(src, srcstride)”而put_hevc_qpel_uni_h_8()中滤波的宏是“QPEL_FILTER(src, 1)”。如此一来就选择了垂直的8个点进行滤波。

put_hevc_qpel_uni_hv_8()

put_hevc_qpel_uni_hv_8()是单向预测中间位置像素插值函数。该函数的定义如下所示。

```cpp
/*
 * 单向预测
 * 中间（hv）滤波像素插值
 *
 * 需要水平滤波和垂直滤波
 *
 */
static void FUNC(put_hevc_qpel_uni_hv)(uint8_t *_dst,  ptrdiff_t _dststride,
                                       uint8_t *_src, ptrdiff_t _srcstride,
                                       int height, intptr_t mx, intptr_t my, int width)
{
    int x, y;
    const int8_t *filter;
    pixel *src = (pixel*)_src;
    ptrdiff_t srcstride = _srcstride / sizeof(pixel);
    pixel *dst          = (pixel *)_dst;
    ptrdiff_t dststride = _dststride / sizeof(pixel);
    int16_t tmp_array[(MAX_PB_SIZE + QPEL_EXTRA) * MAX_PB_SIZE];
    int16_t *tmp = tmp_array;
    int shift =  14 - BIT_DEPTH;

#if BIT_DEPTH < 14
    int offset = 1 << (shift - 1);
#else
    int offset = 0;
#endif

    src   -= QPEL_EXTRA_BEFORE * srcstride;
    filter = ff_hevc_qpel_filters[mx - 1];
    //先水平像素插值
    for (y = 0; y < height + QPEL_EXTRA; y++) {
        for (x = 0; x < width; x++)
            tmp[x] = QPEL_FILTER(src, 1) >> (BIT_DEPTH - 8);
        src += srcstride;
        tmp += MAX_PB_SIZE;
    }

    tmp    = tmp_array + QPEL_EXTRA_BEFORE * MAX_PB_SIZE;
    filter = ff_hevc_qpel_filters[my - 1];

    //处理x*y个像素
    for (y = 0; y < height; y++) {
        for (x = 0; x < width; x++)
            dst[x] = av_clip_pixel(((QPEL_FILTER(tmp, MAX_PB_SIZE) >> 6) + offset) >> shift);
        tmp += MAX_PB_SIZE;
        dst += dststride;
    }
}
```

从源代码可以看出，put_hevc_qpel_uni_hv_8()是“水平+垂直”的结合。这样就完成了中间位置像素插值的工作。

put_hevc_qpel_bi_h_8()

put_hevc_qpel_bi_h_8()是双向预测水平像素插值函数。该函数的定义如下所示。

```cpp
/*
 * 双向预测
 * 水平（Horizontal）滤波像素插值
 * 注：双向预测要求将滤波后的像素叠加到另一部分像素（单向运动补偿得到的像素）上后求平均
 *
 *
 *  A   B   C   D X E   F   G   H
 *
 *
 * 参数：
 * _dst：输出的插值后像素
 * _dststride：输出一行像素数据的大小
 * _src：输入的整像素
 * _srcstride：输入一行像素数据的大小
 *
 * src2：需要叠加的像素。该像素与滤波后的像素叠加后求平均
 *
 * height：像素的宽
 * width：像素的高
 * mx：运动矢量亚像素x方向取值。以1/4像素为基本单位
 * my：运动矢量亚像素x方向取值。以1/4像素为基本单位
 *
 */
static void FUNC(put_hevc_qpel_bi_h)(uint8_t *_dst, ptrdiff_t _dststride, uint8_t *_src, ptrdiff_t _srcstride,
                                     int16_t *src2,
                                     int height, intptr_t mx, intptr_t my, int width)
{
    int x, y;
    pixel        *src       = (pixel*)_src;
    ptrdiff_t     srcstride = _srcstride / sizeof(pixel);
    pixel *dst          = (pixel *)_dst;
    ptrdiff_t dststride = _dststride / sizeof(pixel);
    //ff_hevc_qpel_filters[]是滤波器插值系数数组
    //[0]为1/4像素点插值；[1]为半像素点插值；[2]为3/4像素点插值
    const int8_t *filter    = ff_hevc_qpel_filters[mx - 1];
    //注意和单向预测相比多了“+1”，即在后面代码中多右移一位，实现了“除以2”功能
    int shift = 14  + 1 - BIT_DEPTH;
#if BIT_DEPTH < 14
    int offset = 1 << (shift - 1);
#else
    int offset = 0;
#endif

    //处理x*y个像素
    //注意，在这里使用QPEL_FILTER[]插值后的像素叠加了src2[]然后求平均
    //这里求平均是通过把shift变量加1实现的（等同于除以2）
    for (y = 0; y < height; y++) {
        for (x = 0; x < width; x++)
            dst[x] = av_clip_pixel(((QPEL_FILTER(src, 1) >> (BIT_DEPTH - 8)) + src2[x] + offset) >> shift);
        src  += srcstride;
        dst  += dststride;
        src2 += MAX_PB_SIZE;
    }
}
```

从源代码可以看出，put_hevc_qpel_bi_h_8()的流程和put_hevc_qpel_uni_h_8()的流程基本上是一样的。由于该函数用于双向预测，所以在求结果的时候是和输入像素“求平均”而不是“赋值”。具体的代码中就是通过将滤波结果与src2[x]相加后除以2实现的（除以2是通过在前面代码中将shift加1实现的）。

剩下的几个插值函数的原理基本一样，在这里不再重复叙述。至此有关FFmpeg HEVC解码器中PU解码部分的代码就分析完毕了。

雷霄骅
leixiaohua1020@126.com
http://blog.csdn.net/leixiaohua1020

文章标签： FFmpeg CTU PU 运动补偿 像素内插

个人分类： FFMPEG

所属专栏： FFmpeg