

JVET-00236

AHG17: Making PPS parsing independent of SPS



Rickard Sjöberg, Martin Pettersson, Mitra Damghanian
Ericsson

Background and problem description



- Keeping the PPS parsing independent of the SPS is a desirable property in VVC
 - If PPS parsing depends on decoded SPS syntax, PPSes may need to be stored in coded format until activation
 - This assumes HEVC-like parameter set activation where parameter sets are activated by VCL NAL units
- Two PPS dependencies were identified in JVET-N1001-v8
 1. The number of bits for `pps_virtual_boundaries_pos_x` and `pps_virtual_boundaries_pos_y` depend on the picture size specified in the SPS
 2. The for loop count for parsing brick syntax elements is equal to `NumTilesInPic`. If `uniform_tile_spacing_flag` is equal to 1, then `NumTilesInPic` depends on the picture size specified in the SPS

1: Virtual boundary signaling in VVC



pic_parameter_set_rbsp() {	Descriptor
...	
pps_num_ver_virtual_boundaries	u(2)
for(i = 0; i < pps_num_ver_virtual_boundaries; i++)	
pps_virtual_boundaries_pos_x[i]	u(v)
pps_num_hor_virtual_boundaries	u(2)
for(i = 0; i < pps_num_hor_virtual_boundaries; i++)	
pps_virtual_boundaries_pos_y[i]	u(v)
...	

pps_virtual_boundaries_pos_x[i] is used to compute the value of `PpsVirtualBoundariesPosX[ i ]`, which specifies the location of the i-th vertical virtual boundary in units of luma samples. The number of bits used to represent `pps_virtual_boundaries_pos_x[ i ]` is $\text{Ceil}(\text{Log}_2(\text{pic_width_in_luma_samples}) - 3)$. `pps_virtual_boundaries_pos_x[ i ]` shall be in the range of 1 to $\text{Ceil}(\text{pic_width_in_luma_samples} \div 8) - 1$, inclusive.

The location of the vertical virtual boundary `PpsVirtualBoundariesPosX[ i ]` is derived as follows:

$$\text{PpsVirtualBoundariesPosX}[i] = \text{pps_virtual_boundaries_pos_x}[i] * 8$$

pps_virtual_boundaries_pos_y[i] is used to compute the value of `PpsVirtualBoundariesPosY[ i ]`, which specifies the location of the i-th horizontal virtual boundary in units of luma samples. The number of bits used to represent `pps_virtual_boundaries_pos_y[ i ]` is $\text{Ceil}(\text{Log}_2(\text{pic_height_in_luma_samples}) - 3)$. `pps_virtual_boundaries_pos_y[ i ]` shall be in the range of 1 to $\text{Ceil}(\text{pic_height_in_luma_samples} \div 8) - 1$, inclusive.

The location of the horizontal virtual boundary `PpsVirtualBoundariesPosY[ i ]` is derived as follows:

$$\text{PpsVirtualBoundariesPosY}[i] = \text{pps_virtual_boundaries_pos_y}[i] * 8$$

1: Virtual boundary signaling - proposal



pic_parameter_set_rbsp() {	Descriptor
...	
pps_num_ver_virtual_boundaries	u(2)
for(i = 0; i < pps_num_ver_virtual_boundaries; i++) {	
pps_virtual_boundaries_coarse_x[i]	ue(v)
pps_virtual_boundaries_fine_x[i]	u(6)
}	
pps_num_hor_virtual_boundaries	u(2)
for(i = 0; i < pps_num_hor_virtual_boundaries; i++) {	
pps_virtual_boundaries_coarse_y[i]	ue(v)
pps_virtual_boundaries_fine_y[i]	u(6)
}	
...	

pps_virtual_boundaries_coarse_x[i] is used to compute the value of PpsVirtualBoundariesPosX[i], which specifies the location of the i-th vertical virtual boundary in units of luma samples.

pps_virtual_boundaries_fine_x[i] is used to compute the value of PpsVirtualBoundariesPosX[i].

The location of the vertical virtual boundary PpsVirtualBoundariesPosX[i] is derived as follows:

$$\text{PpsVirtualBoundariesPosX}[i] = \text{pps_virtual_boundaries_coarse_x}[i] * 512 + \text{pps_virtual_boundaries_fine_x}[i] * 8$$

pps_virtual_boundaries_coarse_y[i] is used to compute the value of PpsVirtualBoundariesPosY[i], which specifies the location of the i-th horizontal virtual boundary in units of luma samples.

pps_virtual_boundaries_fine_y[i] is used to compute the value of PpsVirtualBoundariesPosY[i].

The location of the horizontal virtual boundary PpsVirtualBoundariesPosY[i] is derived as follows:

$$\text{PpsVirtualBoundariesPosY}[i] = \text{pps_virtual_boundaries_coarse_y}[i] * 512 + \text{pps_virtual_boundaries_fine_y}[i] * 8$$

2: Uniform tile signaling



Problem (repeated):

- The for loop count for parsing brick syntax elements is equal to NumTilesInPic.
- If uniform_tile_spacing_flag is equal to 1, then NumTilesInPic depends on the picture size specified in the SPS

Proposed solution

- Add a codeword num_tiles_in_pic_minus1 in case uniform_tile_spacing_flag and brick_splitting_present_flag are both equal to 1
- Note that the variable NumTilesInPic is set equal to $(\text{num_tile_columns_minus1} + 1) * (\text{num_tile_rows_minus1} + 1)$
- If uniform_tile_spacing_flag is equal to 0, num_tile_columns_minus1 and num_tile_rows_minus1 are sent in the PPS
- If brick_splitting_present_flag is equal to 0, there is no for loop

pic_parameter_set_rbsp() {	Descriptor
...	
brick_splitting_present_flag	u(1)
if(uniform_tile_spacing_flag && brick_splitting_present_flag)	
num_tiles_in_pic_minus1	ue(v)
for(i = 0; brick_splitting_present_flag && i <= num_tiles_in_pic_minus1 NumTilesInPic; i++) {	
brick_split_flag[i]	u(1)
...	

num_tiles_in_pic_minus1 plus 1 specifies the number of tiles in each picture referring to the PPS. The value of num_tiles_in_pic_minus1 shall be set equal to NumTilesInPic – 1 and when not present, the value of num_tiles_in_pic_minus1 is inferred to be equal to NumTilesInPic – 1.





pic_parameter_set_rbsp() {	Descriptor
pps_pic_parameter_set_id	ue(v)
pps_seq_parameter_set_id	ue(v)
output_flag_present_flag	u(1)
single_tile_in_pic_flag	u(1)
if(!single_tile_in_pic_flag) {	
uniform_tile_spacing_flag	u(1)
if(uniform_tile_spacing_flag) {	
tile_cols_width_minus1	ue(v)
tile_rows_height_minus1	ue(v)
} else {	
num_tile_columns_minus1	ue(v)
num_tile_rows_minus1	ue(v)
for(i = 0; i < num_tile_columns_minus1; i++)	
tile_column_width_minus1[i]	ue(v)
for(i = 0; i < num_tile_rows_minus1; i++)	
tile_row_height_minus1[i]	ue(v)
}	
brick_splitting_present_flag	u(1)
if(uniform_tile_spacing_flag && brick_splitting_present_flag)	
num_tiles_in_pic_minus1	
for(i = 0; brick_splitting_present_flag && i <= num_tiles_in_pic_minus1 NumTilesInPic; i++) {	
brick_split_flag[i]	u(1)
if(brick_split_flag[i]) {	
uniform_brick_spacing_flag[i]	u(1)
if(uniform_brick_spacing_flag[i])	
brick_height_minus1[i]	ue(v)
else {	
num_brick_rows_minus1[i]	ue(v)
for(j = 0; j < num_brick_rows_minus1[i]; j++)	
brick_row_height_minus1[i][j]	ue(v)
}	
}	
}	
}	
}	