OCL Device-Side AVC VME Programmers Manual

Version 1.0

External Revision History

Date	Author	Comment
03/21/2017	Biju George	Version v1.0 for Linux MSS 2017 R3 and Windows 15.46 release

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm%20 Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations

All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Celeron, Intel, Intel logo, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel SpeedStep, Intel XScale, Itanium, Pentium, Pentium Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Intel® Active Management Technology requires the platform to have an Intel® AMT-enabled chipset, network hardware and software, as well as connection with a power source and a corporate network connection. With regard to notebooks, Intel AMT may not be available or certain capabilities may be limited over a host OS-based VPN or when connecting wirelessly, on battery power, sleeping, hibernating or powered off. For more information, see http://www.intel.com/technology/iamt.

64-bit computing on Intel architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

No computer system can provide absolute security under all conditions. Intel® Trusted Execution Technology is a security technology under development by Intel and requires for operation a computer system with Intel® Virtualization Technology, an Intel Trusted Execution Technology-enabled processor, chipset, BIOS, Authenticated Code Modules, and an Intel or other compatible measured virtual machine monitor. In addition, Intel Trusted Execution Technology requires the system to contain a TPMv1.2 as defined by the Trusted Computing Group and specific software for some uses. See http://www.intel.com/technology/security/ for more information.

†Hyper-Threading Technology (HT Technology) requires a computer system with an Intel® Pentium® 4 Processor supporting HT Technology and an HT Technology-enabled chipset, BIOS, and operating system. Performance will vary depending on the specific hardware and software you use. See www.intel.com/products/ht/hyperthreading_more.htm for more information including details on which processors support HT Technology.

Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM) and, for some uses, certain platform software enabled for it. Functionality, performance or other benefits will vary depending on hardware and software configurations and may require a BIOS update. Software applications may not be compatible with all operating systems. Please check with your application vendor.

* Other names and brands may be claimed as the property of others.

Other vendors are listed by Intel as a convenience to Intel's general customer base, but Intel does not make any representations or warranties whatsoever regarding quality, reliability, functionality, or compatibility of these devices. This list and/or these devices may be subject to change without notice.

Contents

Legal Disclaimer 2

Acknowledgements........ 9

Introduction 10

Programming model 12

Intel OpenCL Subgroups........ 12

VME subgroup functions........ 14

VME function phases 16

VME function categories........ 19

Sample OpenCL motion estimation pipeline 20

Restrictions 21

Programming interface 24

Basic definitions 24

Device support 25

Source and reference images 25

Source and Reference image parameters........ 27

Exclusive use of image parameters........ 28

Inline media sampler........ 29

IME functions 30

Initialization phase 32

Reference window configuration phase 32

Cost configuration phase 38

Miscellaneous property configuration phase 46

Evaluation and result processing phase........ 47

REF functions 59

Initialization & Configuration phase 63

Cost configuration phase 65

Miscellaneous property configuration phase 66

Evaluation and result processing phase........ 67

SIC functions 71

Initialization phase 74

Operation configuration phase 74

Cost configuration phase 79

Miscellaneous property configuration phase 86

Evaluation and result processing phase 88

VME with no macroblock dependencies (fully parallel VME) 94

Processing one macroblock per workgroup 94

Processing a column of macroblocks per workgroup 95

VME with macroblock dependencies (wavefront parallelism) 96

Wavefront processing of macroblocks 98

Software Scoreboarding 102

Application of OpenCL for HEVC encode pipeline 108

References 110

Figures

FIGURE 1: GEN8 ARCHITECTURAL DIAGRAM....10
FIGURE 2: INTEL OPENCL SUBGROUPS PROGRAMMING MODEL....13
FIGURE 3: INTEL VME SUBGROUPS PROGRAMMING MODEL....14
FIGURE 4: GENERIC VME FUNCTION PHASES FLOWCHART....17
FIGURE 5: VME FUNCTION PHASES USAGE EXAMPLE....18
FIGURE 6: INTEL OPENCL PIPELINE EXAMPLE USING DEVICE-SIDE VME....20
FIGURE 7: IME OPERATION....30
FIGURE 8: IME FUNCTION PHASE FLOW....31
FIGURE 9: SINGLE REFERENCE REGION OFFSETTING....33
FIGURE 10: EXHAUSTIVE SEARCH PATH FOR A 48X40 SEARCH WINDOW....34
FIGURE 11: DIAMOND SEARCH PATH (PRE-DETERMINED PART) FOR A 48X40 SEARCH WINDOW....34
FIGURE 12: EXHAUSTIVE SEARCH PATH FOR A 32X32 SEARCH WINDOW....36
FIGURE 13: DIAMOND SEARCH PATH (PRE-DETERMINED PART) FOR A 32X32 SEARCH WINDOW....36
FIGURE 14: DUAL-REFERENCE SEARCH REGION OFFSETTING....37
FIGURE 15: INTER DISTORTION CALCULATION....39
FIGURE 16: MULTI-REFERENCE SCALING FACTOR BY REFERENCE IDENTIFIER....40
FIGURE 17: SAMPLE MODEL FOR CUSTOM INTER SHAPE COSTS....43
FIGURE 18: COST FUNCTION DESCRIPTION....45
FIGURE 19: SAMPLE MODEL FOR CUSTOM MOTION VECTOR COST TABLE....46
FIGURE 20: IME MOTION-VECTOR WORK-ITEM MAPPING....50
FIGURE 21: STREAMIN/STREAMOUT IME....54
FIGURE 22: SUB-PIXEL REFINEMENT....60
FIGURE 23: FME 4-TAP FILTER....60
FIGURE 24: BIDIRECTIONAL MOTION ESTIMATION....61
FIGURE 25: REF FUNCTION PHASE FLOW CHART....62
FIGURE 26: INTRA-PREDICTION MODES FOR 4X4 AND 8X8....72
FIGURE 27: SIC FUNCTION PHASES FLOW CHART....73
FIGURE 28: INTRA LUMA DISTORTION CALCULATION....79
FIGURE 29: INTRA CHROMA DISTORTION CALCULATION....80
FIGURE 30: INTRA LUMA SHAPE PENALTY CONFIGURATION....81
FIGURE 31: SAMPLE MODEL FOR INTRA SHAPE PENALTY....82
FIGURE 32: Z-ORDER LAYOUT....83
FIGURE 33: SAMPLE MODEL LUMA MODE PENALTY....85
FIGURE 34: INTRA CHROMA MODE COST CONFIGURATION....86
FIGURE 35: USE OF FRAME-LEVEL VME IN GENERAL ENCODE PIPELINE....94
FIGURE 36: OPENCL HOST ENQUEUE FOR FRAME-LEVEL VME WITH NO MACROBLOCK DEPENDENCIES....95
FIGURE 37: AVC MACROBLOCK DEPENDENCIES (26 DEGREE WAVEFRONT)....96
FIGURE 38: VME WITH MACRO-BLOCK DEPENDENCIES IN GENERAL ENCODE PIPELINE....97
FIGURE 39: 45 DEGREE WAVEFRONT PROCESSING....98
FIGURE 40: WORKGROUP ASSIGNMENTS FOR FRAME....99
FIGURE 41: SCOREBOARD REPRESENTATION....102

FIGURE 42: SCOREBOARD BLOCKING AND SIGNALING 107
FIGURE 43: SINGLE STAGE GPU ACCELERATED CPU ENCODER 108
FIGURE 44: HYBRID TWO STAGE GPU ENCODER 109

Tables

TABLE 1: TERMS AND DEFINITIONS 25
TABLE 2: SINGLE-REFERENCE SEARCH WINDOW CONFIGURATIONS 34
TABLE 3: DUAL-REFERENCE SEARCH WINDOW CONFIGURATIONS 36
TABLE 4: INTER PACKED SHAPE PENALTY FORMAT 42
TABLE 5: INTER MOTION VECTOR COST PRECISION UNITS AND RANGES 44
TABLE 6: INTER MACROBLOCK BLOCK REFERENCE IDENTIFIERS FORMAT 57
TABLE 7: PACKED REFERENCE BLOCK FIELD POLARITIES 58
TABLE 8: INTRA PACKED SHAPE PENALTY FORMAT 81
TABLE 9: INTRA PACKED LUMA NEIGHBOR MODE FORMAT 83
TABLE 10: INTRA PACKED NON-DC PENALTY FORMAT 83
TABLE 11: CHROMA MODE BASE PENALTY SCALE FACTORS 86
TABLE 12: PACKED SAD COEFFICIENT THRESHOLDS FORMAT 87

Acknowledgements

This manual would not have been possible without the advice and previous related work of Raghukrishnan Embar, Ryan Lei, James Holland, Ben Ashbaugh, Saimanohara Alapati, and Jay Patel, and development of sample applications by Sriram Murali, Atul Divekar, and Kalyan Kaipa. Thanks to all of them.

Introduction

The Intel GEN[1] architecture has a Video Motion estimation (VME) unit which accelerates a set of motion estimation operations on a given source macroblock. It computes the motion vectors, intra prediction angles and macroblock partitioning combination that best describes the transformation to the source macroblock, from blocks in one or more previous reference pictures (inter-estimation), or from other blocks in the same source picture (intra-estimation). It does this by searching for spatial and temporal patterns on the current and various forward and backward reference pictures.

The VME unit is part of the media sampler block as shown in the GEN8 (Broadwell) architectural diagram below.

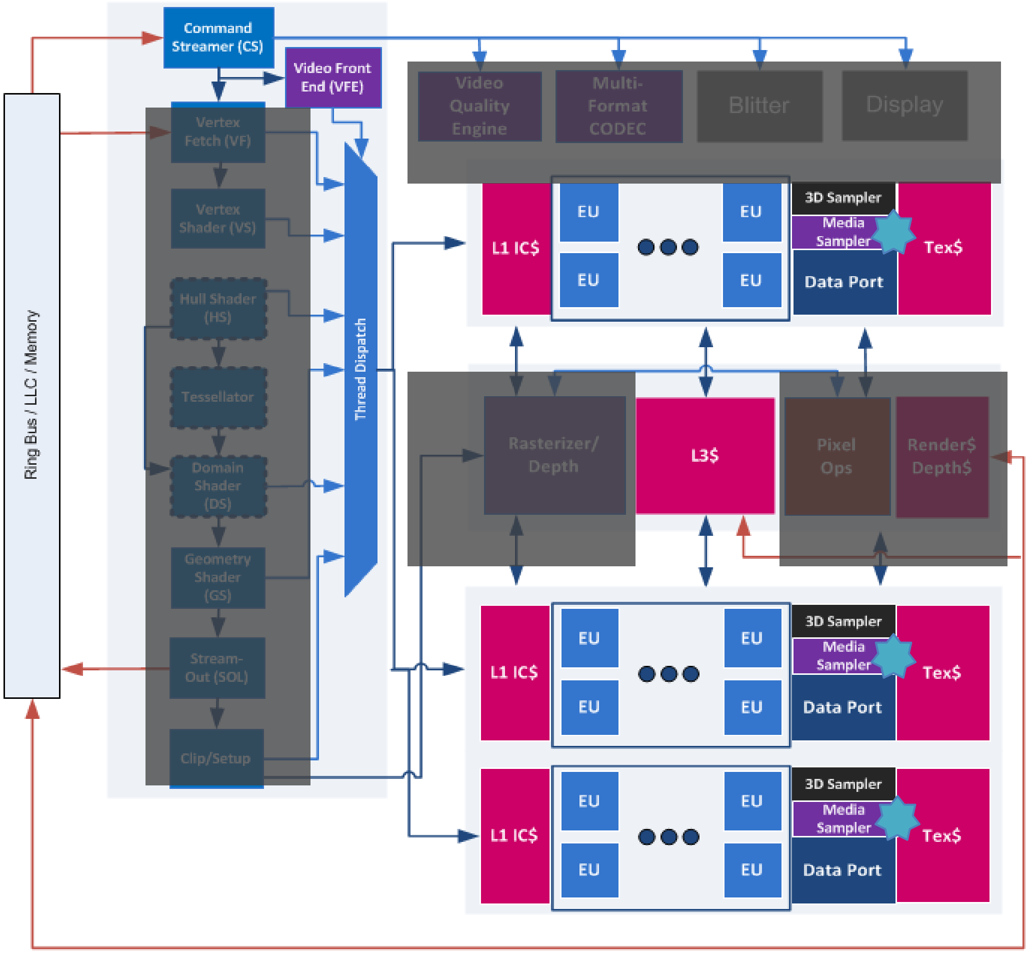


Figure 1: GEN8 architectural diagram

Each GEN sub-slice contains an array of multi-threaded SIMD execution units (EUs) with its own local thread dispatcher unit and supporting instruction caches, along with its fixed function assets of the

[1] GEN refers to Intel on-die integrated processor graphics.

media and 3D sampler, and the data port for memory management. The global thread dispatcher will perform the load balancing by attempting to distribute the workload evenly across the sub-slices.

The media sampler is accessible from SIMD kernel threads, executing on the execution units (EUs) of a subslice, for accelerating media operations just as the 3D texture sampler is used for accelerating 3D texture sampling operations. Therefore motion estimation on GEN is performed by executing SIMD kernel code on the EUs which leverages the VME unit for acceleration of specific motion estimation operations.

Sub-slices are further clustered into slices which form the basis of the scalable GEN design. VME units are replicated in each sub-slice being a sub-slice local asset. In the GEN8 and GEN9 (Skylake) architectures three sub-slices form a slice. Different product configuration may have a different number of slices.

A detailed description of the GEN8 and GEN9 compute architectures are described in [1] and [2] respectively.

The Intel OpenCL *cl_intel_device_side_avc_motion_estimation* extension described in [4] provides a fine-grained interface to access the hardware supported AVC VME[2] functionality from OpenCL kernels. It describes the specification of low-level built-in functions that facilitate the programming of the VME unit to evaluate specific motion estimation operations, and is callable from OpenCL kernels executing on the EUs.

This document will elaborate on the VME operations exposed by built-in functions described in [4] explain its programming model, describe its usage and provide guidelines for writing OpenCL motion estimation kernels.

[2] The definition of HEVC VME built-in functions is intended to be part of another extension for future GEN architectures.

Programming model

Intel OpenCL Subgroups

In the OpenCL parallel programming model a kernel is defined for a work-item, i.e. each instance of a kernel is a work-item.

OpenCL kernels execute in parallel over a specified global n-dimensional range (NRange) of work-items called the global work size, where 'n' can be up to 3. Each work-item in the global NDRange gets assigned a unique n-dimensional global identifier that can be queried using *get_global_id(dim)*.

Work-items get grouped into subgroups. Each work-item within the subgroup gets assigned an additional one-dimensional local identifier that is unique within the subgroup, and can be queried using *get_sub_group_local_id()*. The exact implementation of subgroups may differ across platform vendors. For the Intel GEN platform, a subgroup maps onto a SIMD kernel hardware thread. Intel OpenCL Subgroups is described in [3]. Subgroups are one dimensional and its size can optionally be fixed by the kernel attribute *intel_reqd_sub_group_size(size)* where '*size*' is either 8, 16 or 32, otherwise the compiler decides the optimal subgroup size based on analysis of the kernel code.

Work-items within a subgroup may cooperate with one another using the sub-group shuffle built-in functions without using shared local memory or barriers and thus achieve better performance by keeping data in registers.

Subgroups are used to define block based collective operations where the collective work-items in a subgroup perform a single block operation. The results of the single block operation will get mapped back into the individual work-items of the subgroup by the convention described in the description of the API for the block based operation.

An important restriction for subgroup block based collective functions is that they cannot be used in divergent control flow; i.e. if used in conditional branches or loops, the branch or loop condition must have the same value for all work-items within a subgroup, else the behavior is undefined.

Subgroups get grouped into workgroups. Each subgroup gets assigned a unique one-dimensional local identifier that is unique within the workgroup and can be queried using *get_sub_group_id()*. Each work-item within a workgroup gets assigned an additional n-dimensional workgroup local identifier that is unique within the workgroup and can be queried using *get_local_id(dim)*, as local workgroups can be specified over an n-dimensional space. Work-items within a workgroup may cooperate with one another using shared local memory, fences, atomics, and barriers for synchronization. Note that the cooperation mechanism for work-items within a workgroups is less efficient than using subgroups.

Figure 2 shown below is an example with a global one-dimensional NDRange of size (8, 1), a local one-dimensional workgroup size of (4, 1) and a subgroup size of 8[3] with only the only first 2 work-items in

[3] Subgroups sizes allowed are 8, 16 or 32 as supported by the SIMD kernel threads in GEN.

the subgroup enabled. Note that the sizes chosen in the figure are only for pictorial convenience and not recommended for performance considerations (only 2 of the SIMD channels of a subgroup capable of performing SIMD8 computations are efficiently utilized resulting in the loss of compute bandwidth).

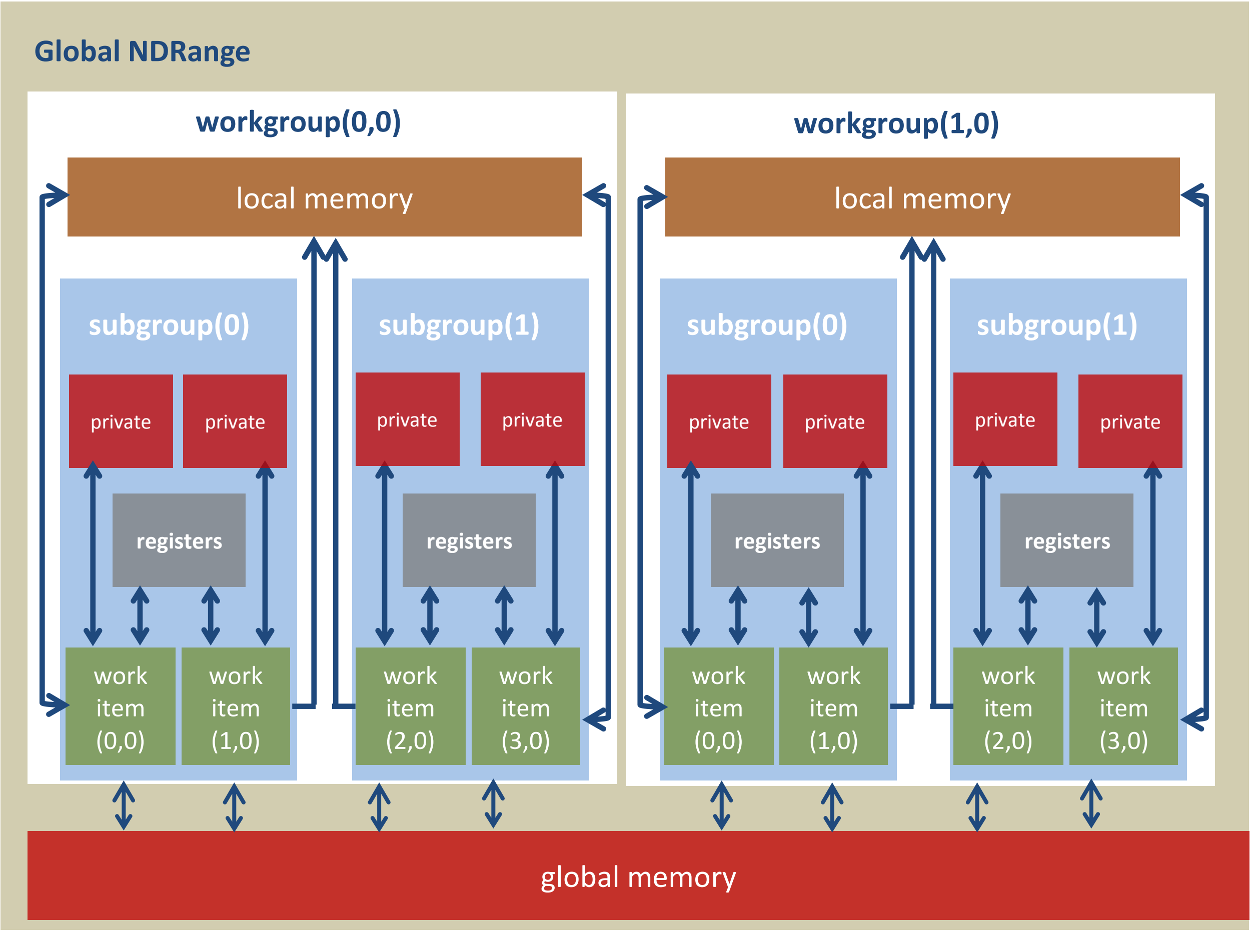
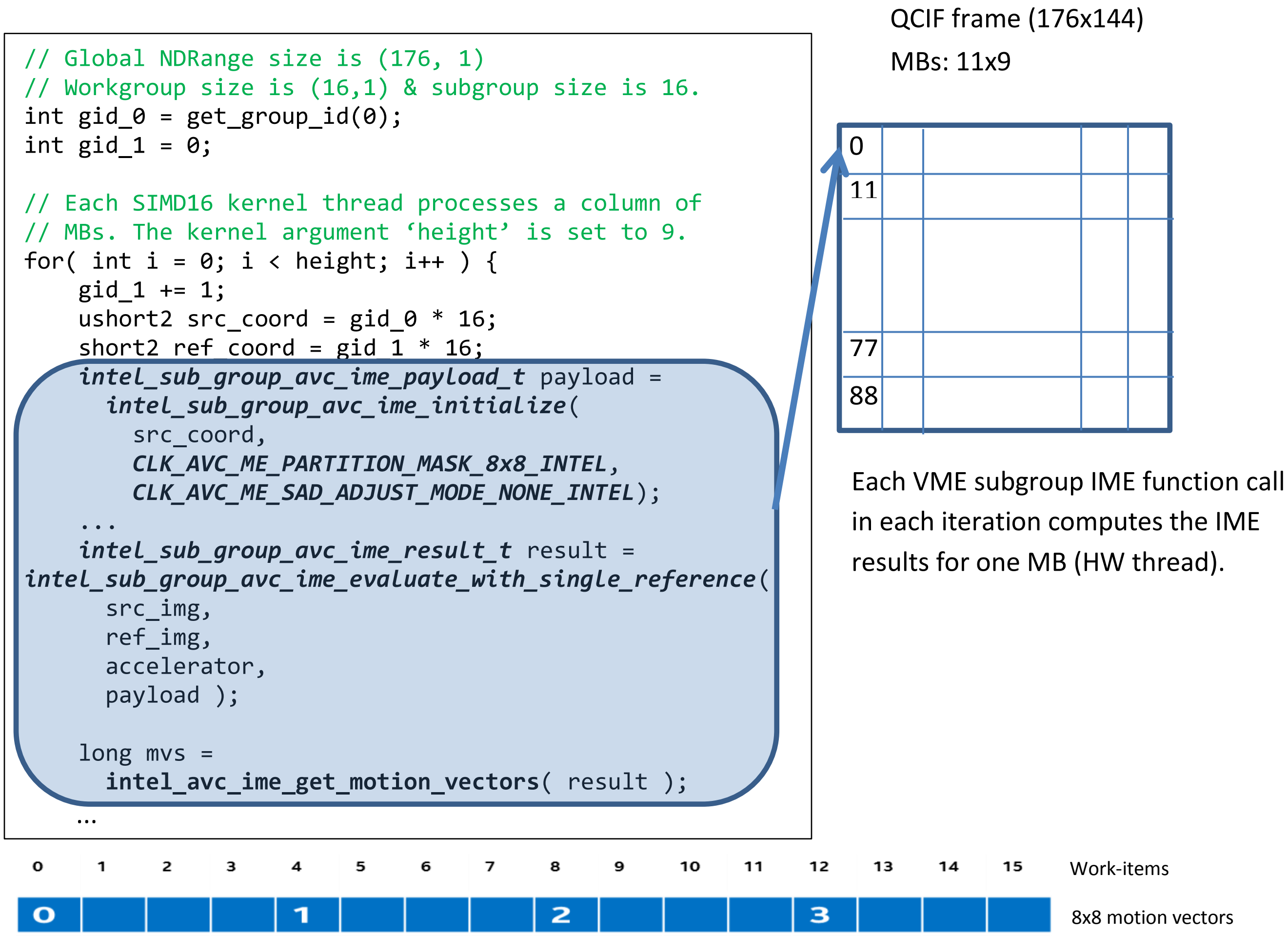


Figure 2: Intel OpenCL subgroups programming model

VME subgroup functions

VME operations are block operations operating on a 16x16 macroblock. The device-side VME built-in functions, being a block API, are subgroup collective functions. The use of VME built-in functions in a kernel generate an implicit kernel attribute *intel_reqd_sub_group_size(16)* which forces the sub-group size as 16.

When calling a VME built-in function in a kernel, 16 work-items collectively perform a single specific VME operation on a 16x16 macroblock[4] and its result components get mapped to specific work-items as described in the specification for the specific result extracting subgroup built-in functions.



```
// Global NDRange size is (176, 1)
// Workgroup size is (16,1) & subgroup size is 16.
int gid_0 = get_group_id(0);
int gid_1 = 0;

// Each SIMD16 kernel thread processes a column of
// MBs. The kernel argument ‘height’ is set to 9.
for( int i = 0; i < height; i++ ) {
    gid_1 += 1;
    ushort2 src_coord = gid_0 * 16;
    short2 ref_coord = gid_1 * 16;
    intel_sub_group_avc_ime_payload_t payload =
      intel_sub_group_avc_ime_initialize(
        src_coord,
        CLK_AVC_ME_PARTITION_MASK_8x8_INTEL,
        CLK_AVC_ME_SAD_ADJUST_MODE_NONE_INTEL);
    ...
    intel_sub_group_avc_ime_result_t result =
intel_sub_group_avc_ime_evaluate_with_single_reference(
      src_img,
      ref_img,
      accelerator,
      payload );

    long mvs =
      intel_avc_ime_get_motion_vectors( result );
    …
```

Each VME subgroup IME function call in each iteration computes the IME results for one MB (HW thread).

Figure 3: Intel VME subgroups programming model

The programming model for the VME built-in functions is based on first creating and initializing a VME operation payload using initialization built-in functions in kernels executing on the EUs, which is then configured for a set of specific sub-functions to compute using a set of configuration built-in functions, and then finally evaluating the configured VME operation payload using an evaluation built-in functions

[4] It must be noted that this is different from typical OpenCL functions which are defined for a single work-item.

which issues it to the VME unit. The payload and evaluation result are opaque objects that are kept in the subgroup kernel thread’s registers. A set of result processing built-in subgroup functions are called in order to extract the various component results from the evaluation result, such as motion vectors or distortions. These built-in function specifications describe the layout of the result components among the subgroups work-items.

VME function phases

A set of ordered phases of built-in functions need to be called to evaluate a particular VME operation. The ordered phases are described below. Some phases may not be present for certain operations, but if present it must be called in the correct order, otherwise the behavior is undefined.

1. *Initialization (required phase)*
   This creates and initializes the payload for the VME operation.

2. *Operation configuration (required phase, if present for the specific operation)*
   This configures important parameters for the VME operation that were not set in the initialization phase in the payload.

3. *Inter cost configuration (optional phase)*
   This configures the heuristics designed in the VME unit to model the rate-distortion optimization (RDO) for inter estimation with the objective to minimize the bits in encoding.

4. *Intra cost configuration (optional phase)*
   This configures the heuristics designed in the VME unit to model the rate-distortion optimization (RDO) for intra estimation with the objective to minimize the bits in encoding.

5. *Miscellaneous property configuration (optional phase)*
   This configures miscellaneous operation sub-functions in the VME operation payload.

6. *Evaluation (required phase)*
   This phase performs the evaluation of the VME operation using the payload configured in the previous phases by issuing it to the VME unit for processing.

7. *Result processing (required phase)*
   The phase performs the extraction of various motion estimation operation component results from the result of the evaluation phase. The result components may be distributed among multiple work-items. The results components on one VME operation may be used as input to another VME operation at phase (1) or phase (5).

In most cases the compiler will be able to eliminate any redundant moves that may appear to result from the built-in function definitions.

The following figure demonstrates the ordered phases of operations to evaluate a VME operation.

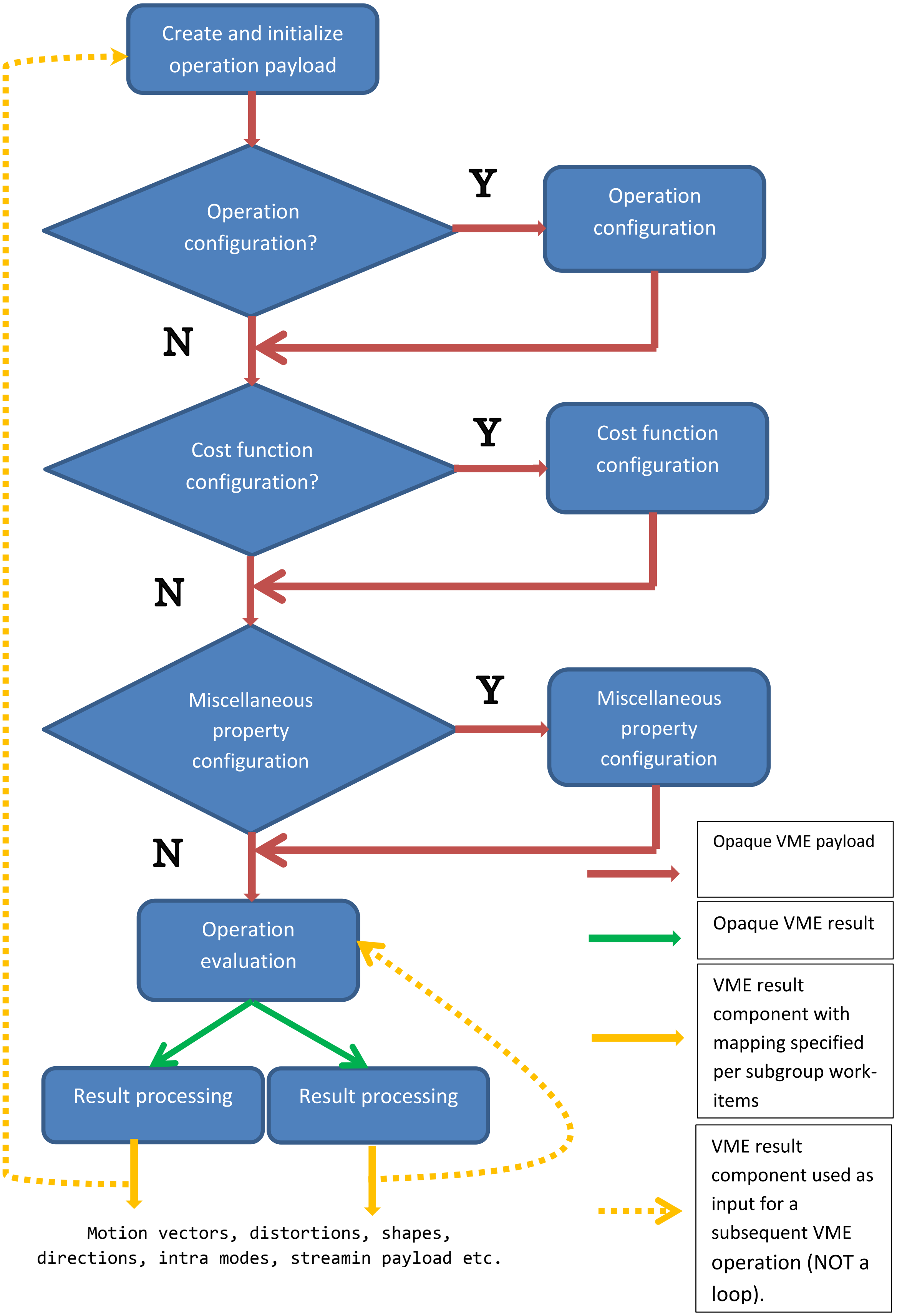
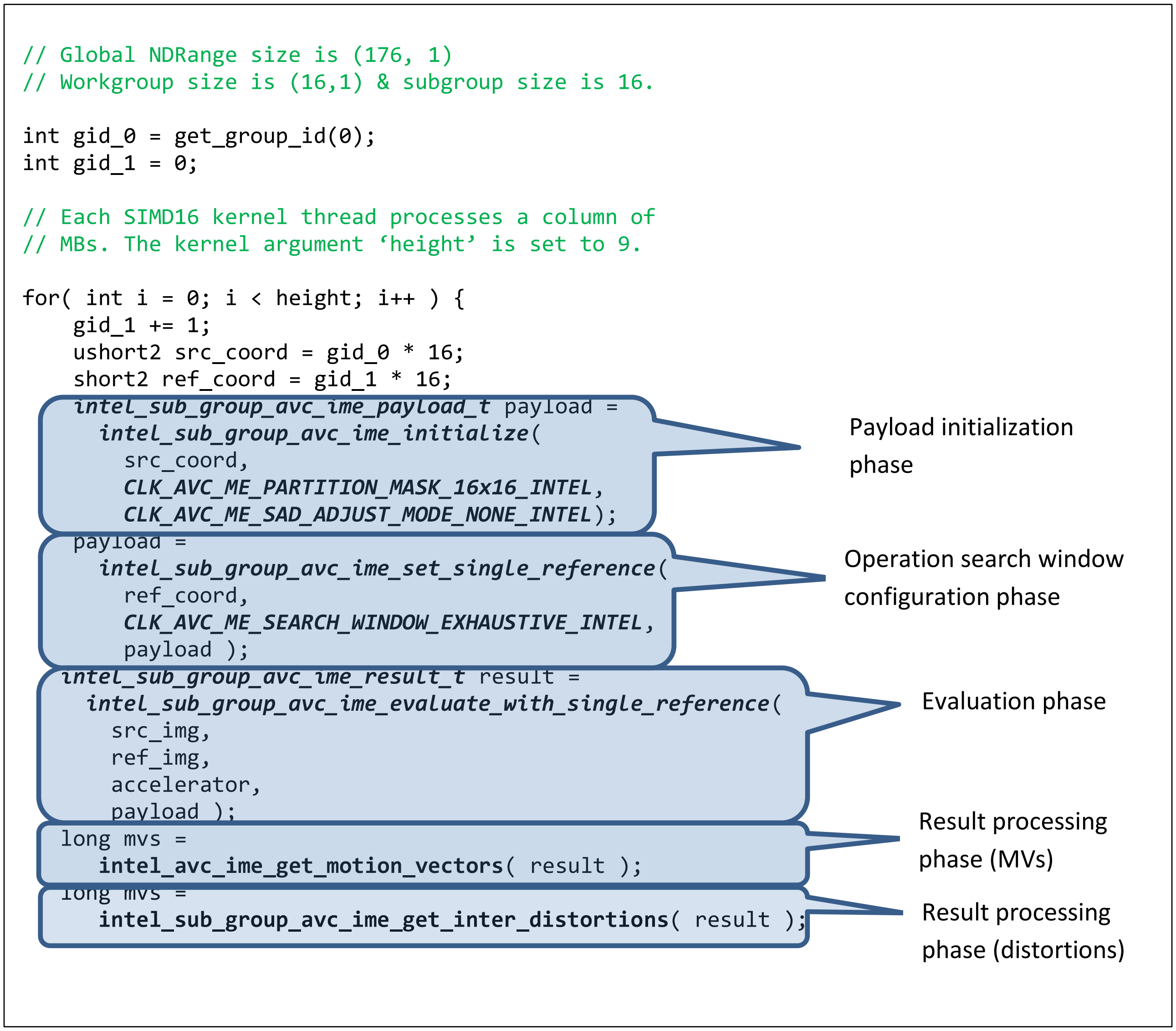

Create and initialize operation payload
Operation configuration?
Y
Operation configuration
N
Cost function configuration?
Y
Cost function configuration
N
Miscellaneous property configuration
Y
Miscellaneous property configuration
N
Operation evaluation
Result processing
Result processing
Motion vectors, distortions, shapes, directions, intra modes, streamin payload etc.
Opaque VME payload
Opaque VME result
VME result component with mapping specified per subgroup work-items
VME result component used as input for a subsequent VME operation (NOT a loop).

Figure 4: Generic VME function phases flowchart

The following figure shows an example of the usage of the built-in IME function phases.



```
// Global NDRange size is (176, 1)
// Workgroup size is (16,1) & subgroup size is 16.

int gid_0 = get_group_id(0);
int gid_1 = 0;

// Each SIMD16 kernel thread processes a column of
// MBs. The kernel argument 'height' is set to 9.

for( int i = 0; i < height; i++ ) {
    gid_1 += 1;
    ushort2 src_coord = gid_0 * 16;
    short2 ref_coord = gid_1 * 16;
    intel_sub_group_avc_ime_payload_t payload =
      intel_sub_group_avc_ime_initialize(
        src_coord,
        CLK_AVC_ME_PARTITION_MASK_16x16_INTEL,
        CLK_AVC_ME_SAD_ADJUST_MODE_NONE_INTEL);
    payload =
      intel_sub_group_avc_ime_set_single_reference(
        ref_coord,
        CLK_AVC_ME_SEARCH_WINDOW_EXHAUSTIVE_INTEL,
        payload );
  intel_sub_group_avc_ime_result_t result =
    intel_sub_group_avc_ime_evaluate_with_single_reference(
      src_img,
      ref_img,
      accelerator,
      payload );
  long mvs =
      intel_avc_ime_get_motion_vectors( result );
  long mvs =
      intel_sub_group_avc_ime_get_inter_distortions( result );
```

Payload initialization phase
Operation search window configuration phase
Evaluation phase
Result processing phase (MVs)
Result processing phase (distortions)

Figure 5: VME function phases usage example

Note: Pass-by-reference is not supported for VME built-in functions since it is based on OpenCL 1.2, and thus the updated payload returned by each function phase must be used as input to the subsequent phase. See example below.

intel_sub_group_avc_ime_set_single_reference(
refCoord, CLK_AVC_ME_SEARCH_WINDOW_EXHAUSTIVE_INTEL, ***payload****);* ***// BAD : Dead code***
payload *= intel_sub_group_avc_ime_set_single_reference(*
refCoord, CLK_AVC_ME_SEARCH_WINDOW_EXHAUSTIVE_INTEL, ***payload****);* ***// GOOD: Updates payload***

VME function categories

A set of built-in functions are defined for each of the major operations of the VME hardware. The earlier mentioned ordered phase of built-in functions needs to be called for evaluating each of the major VME operations. The major operations of the VME hardware are summarized as follows.

Integer motion estimation (IME)

Perform integer motion estimation on a given source macroblock in a source image and a single or dual reference window in a reference image to determine the best integer motion vectors and their associated distortions, and the best macroblock shape partitioning combination.

Motion estimation refinement (REF)

Perform refinement operations on the results of IME. The two sub-operations are:

- ***Fractional motion estimation (FME)***
  Perform sub-pixel refinement on the results of an IME operation. Half-pixel and quarter-pixel refinements are performed to determine the best sub-pixel motion vectors and their associated distortions.
- ***Bidirectional motion estimation (BME)***
  Perform bidirectional refinement on the results of an IME operation using two reference images to check if the bidirectional mode using two references yields lesser distortions. An FME can optionally be performed implicitly as part of a bidirectional refinement.

Skip and Intra check (SIC)

This performs the following two sub-operations.

- ➢ ***Skip check (SKC)***
  Compute the pixel distortion of a user-specified shape and motion vector combination. VME fetches necessary pixels, performs fractional and bidirectional filtering (as necessary), and then computes the SAD/HAAR between the derived reference and source. The skip decision can optionally be enhanced to include a 4x4 forward transform, the results of which are compared against a user specified threshold to emulate the effects of the forward quantization zeroing effect.
- ➢ ***Intra prediction estimation (IPE)***
  Perform intra estimation on a given source macroblock to determine the best intra prediction modes and the best shape partitioning combination.

The GEN VME unit has two dedicated pipelines that can execute in parallel. The first is the IME pipe which is dedicated for the more compute intensive IME operations, and the second one is the CRE (check and refinement unit) pipe for all other VME operations.

Sample OpenCL motion estimation pipeline

Using the different VME function categories, OpenCL codec developers can create varied motion estimation pipelines based on their custom algorithms. A possible pipeline is shown in the figure below.

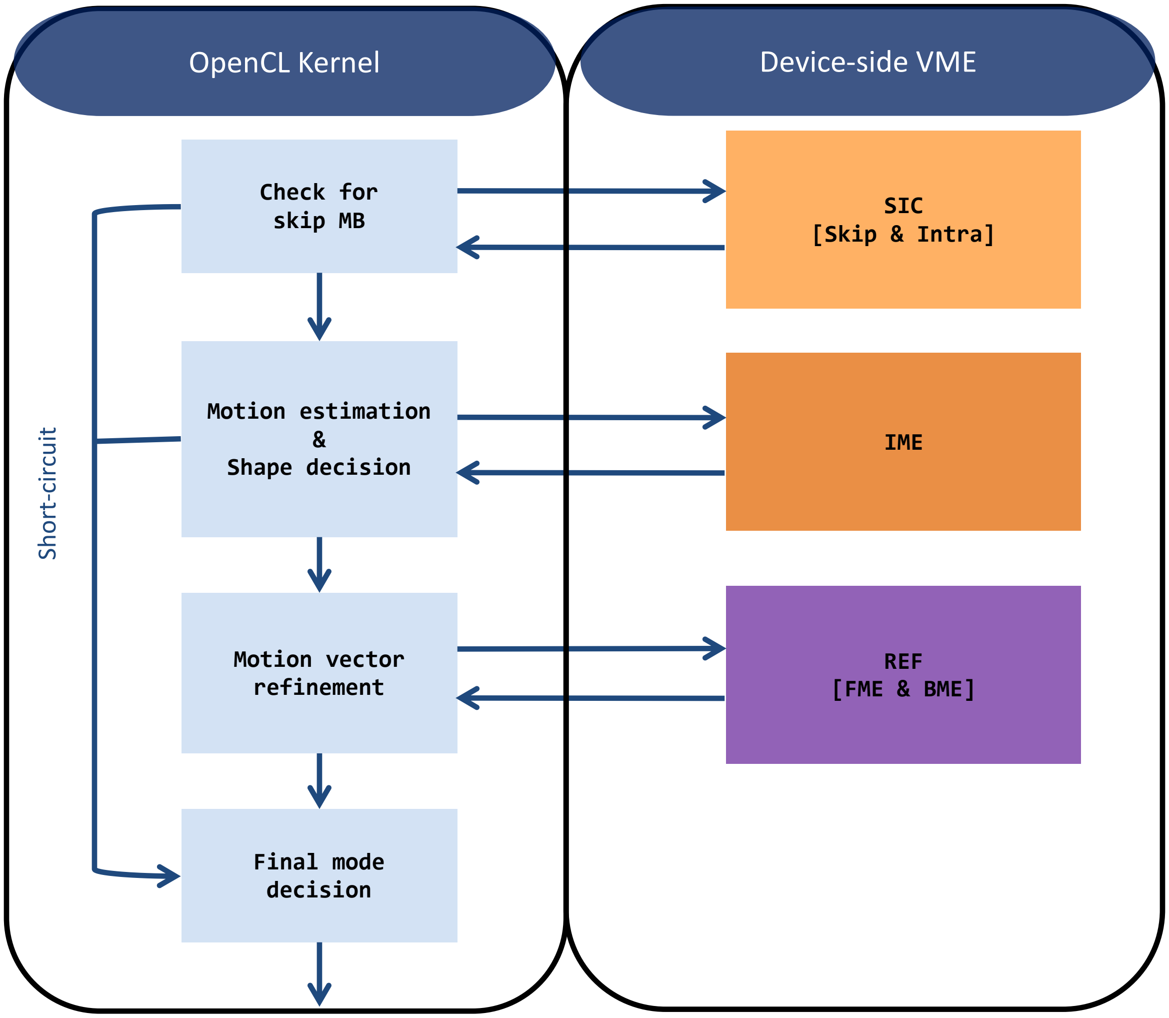


Figure 6: Intel OpenCL pipeline example using device-side VME

Restrictions

1. VME built-in functions are subgroups block functions and therefore cannot be used under divergent control flow.

```
long mvs_8x8 = 0;

// Cannot call VME functions in divergent flow!
// Not all work-items in the subgroup have the same branch condition.
if( ( get_sub_group_local_id() % 4 ) == 0 )
{
  mvs_8x8 = intel_avc_ime_get_motion_vectors( result );
}
```

Illegal

```
long mvs = 0;
long mvs_8x8 = 0;
mvs = intel_avc_ime_get_motion_vectors( result );

if( ( get_sub_group_local_id() % 4 ) == 0 )
{
  mvs_8x8 = mvs;
}
```

Legal

Example 1: VME subgroup function call divergent flow restriction.

2. On GEN8, there is a known hardware issue which effectively disallows texture (3D) sampler accesses when the VME unit is also being used. This prevents most OCL core language image read operations in VME kernels as they use the texture sampler. This issue is resolved in GEN9. In order to read tiled images in VME kernels, media blocks reads as defined in [3] and [5**Error! Reference source not found.**] may be used. In general, using media block reads are more performant than using texture sampler reads. Most VME built-in functions implicitly read the pixels to be processed from the source and reference images, and so additional explicit reads are generally not required except in the case of reading the edge pixels for intra prediction estimation operations.

```
 __kernel void  vme_custom(
    image2d_t src_image,
    image2d_t ref_image,
    image2d_t src_intra_image
 )
{
  sampler_t tex_sampler = CLK_NORMALIZED_COORDS_FALSE;
  sampler_t vme_sampler = CLK_AVC_ME_INITIALIZE_INTEL;
  ...
  intel_sub_group_avc_ime_result_t result =
    intel_sub_group_avc_ime_evaluate_with_single_reference(
      src_img, ref_img, vme_sampler, payload );
  // Cannot use 3D sampler along with VME sampler for GEN8!
  uint data = read_imageui( src_intra_image, tex_sampler, srcCoord ).x;
}
```

Illegal for GEN8; Legal for GEN9

```
 __kernel void  vme_custom(
    image2d_t src_image,
    image2d_t ref_image,
    image2d_t src_intra_image
 )
{
  sampler_t tex_sampler = CLK_NORMALIZED_COORDS_FALSE;
  sampler_t vme_sampler = CLK_AVC_ME_INITIALIZE_INTEL;
  ...
  intel_sub_group_avc_ime_result_t result =
    intel_sub_group_avc_ime_evaluate_with_single_reference(
      src_img, ref_img, vme_sampler, payload );

  // Use a media block read to read contiguous data. Note media block
  // reads return contiguous data across consecutive work-items within
  // the subgroup.
  uint data = intel_sub_group_media_block_read_ui(
      srcCoord, 16 1, src_intra_image );
}
```

Legal

Example 2: Restriction on using texture sampler with VME media sampler and workaround

A device query with the enumeration *CL_DEVICE_AVC_ME_SUPPORTS_TEXTURE_SAMPLER_USE_INTEL*, as described in [4] is provided to check if mixing texture sampler and media sampler accesses is supported by the device.

```
cl_device_id device_id;
...
cl::Device device  = cl::Device( device_id );
clRetainDevice( device_id );
cl_bool deviceSupportsMixedSamplers = false;
cl_bool enumExists =
   device.getInfo<cl_bool>(CL_DEVICE_AVC_ME_SUPPORTS_TEXTURE_SAMPLER_USE_INTEL,
                           &deviceSupportsMixedSamplers);
deviceSupportsMixedSamplers &= ( enumExists == CL_SUCCESS );
if ( deviceSupportsMixedSamplers ) {
    …
}
```

Example 3: Check for devices supporting mixed uses of texture and media sampler

3. On GEN8 and GEN9 there is a known hardware issue which prevents pre-emption for VME kernels. Therefore for these platforms VME operations can only be pre-empted at a frame-level. The intended use of OCL device-side VME is for server-side applications where this is not expected to be an issue.

   A device query with the enumeration *CL_DEVICE_AVC_ME_SUPPORTS_PREEMPTION_INTEL*, as described in [4] is provided to check if mixing texture sampler and media sampler accesses is supported by the device.

```
cl_device_id device_id;
...
cl::Device device  = cl::Device( device_id );
clRetainDevice( device_id );
cl_bool deviceSupportsPremption = false;
cl_bool enumExists =
   device.getInfo<cl_bool >(CL_DEVICE_AVC_ME_SUPPORTS_PREEMPTION_INTEL,
                            &deviceSupportsPremption);
deviceSupportsPremption&= ( enumExists == CL_SUCCESS );
if (deviceSupportsPremption) {
    …
}
```

Programming interface

Basic definitions

Macro-block (MB)	An image is partitioned into macro-blocks of size 16x16 pixels. It is the basic unit of processing for AVC video motion estimation operations.
Shape	A MB may be partitioned into sub-blocks of one of the major shapes. A sub-block with an 8x8 major shape may be further independently partitioned into sub-blocks of one of the minor shapes. It is represented by pre-defined shape enumeration values.
Major Shape	Shapes of 16x16, 16x8, 8x16, or 8x8 partitions of a MB. A 16x16 major shape merely indicates that the MB was not further partitioned.
Minor Shape	Shapes of 8x8, 8x4, 4x8, or 4x4 sub-partitions of an 8x8 partition. A 8x8 minor shape merely indicates that the 8x8 major partition was not further sub-partitioned.
Block	A sub-block of a MB with one of the major or minor shapes.
Reference image	An image from the previously decoded buffer, from which motion estimation predictions are made.
Forward reference image	A reference image from the L0 list as per the AVC specification.
Backward reference image	A reference image from the L1 list as per the AVC specification.
Source image	The current image for which motion estimation predictions are made.
Motion Vector (MV)	A 2D vector used for inter motion estimation that provides an offset from the top left corner of a block in the source image to the top left corner a block in the reference image. Generally it is used to represent the best match of a block in the reference image to a block in the source image. The best match is determined as the block minimizing the distortion. It is represented by a pair of signed 16-bit integer values. MVs are specified in quarter pixel resolution with the 2 LSBs representing the fractional part of the offset. It is represented by a pair of signed 16-bit integers.
Sum Of Absolute Difference (SAD)	The sum of absolute differences of every full/sub-pixel location in the source block w.r.t every corresponding full/sub pixel in the reference block as specified by a given MV. The sum of absolute differences may be optionally Haar transform adjusted (to be used

	as a coarse estimation of the integer transform). It is represented by an unsigned 16-bit integer value.
U4U4 Byte Format	Represents a value of (B<<S), where B, called base, is the 4 bit LSB of the byte and S, called shift, is the 4 bit MSB of the byte.
Luma	Luma refers to either the Y-plane of a NV12 image or a regular image with the image_channel_order and image_data_type restricted as CL_R and CL_UNORM_INT8.
Chroma	Chroma refer to the UV-plane of a NV12 image.

Table 1: Terms and Definitions

Device support

The VME built-in functions can be used by any GEN device that supports the *cl_intel_device_side_avc_motion_estimation* extension. This includes GEN8 and subsequent architectures. This extension is based on OpenCL 1.2.

```
cl_device_id device_id;
...
cl::Device device  = cl::Device( device_id );
clRetainDevice( device_id );
std::string ext = device.getInfo< CL_DEVICE_EXTENSIONS >();
if (string::npos == ext.find("cl_intel_device_side_avc_motion_estimation"))
{
  throw Error("Error,  AVC device-side motion estimation extensions not supported by device!");
 }
```

Example 4: Checking for device support for extension

Source and reference images

The source and reference images may be either 8-bit luma images, or native NV12 images. Luma refers to either the Y-plane of a NV12 image or a regular image with the image_channel_order and image_data_type restricted as CL_R and CL_UNORM_INT8.

If chroma based intra estimation operations are being performed, then native NV12 source images are required, otherwise either 8-bit luma or native NV12 images may be used. Support for native NV12 images is provided by the Intel OpenCL extension cl_intel_planar_yuv as described in [5].

For 8-bit luma images, the channel orders and channel data types are restricted as (CL_R, CL_UNORM_INT8). The host program is responsible for populating the tiled image either using the clEnqueueWriteImage function or its wrapper function enqueueWriteImage, or using the clEnqueueMapImage or its wrapper function enqueueMapImage.

```
cl::ImageFormat imageFormat(CL_R, CL_UNORM_INT8);
cl::Image2D ref_image(context, CL_MEM_READ_ONLY, imageFormat, width, height, 0, 0);
cl::Image2D src_image(context, CL_MEM_READ_ONLY, imageFormat, width, height, 0, 0);
cl::size_t<3> origin; // Init to 0 in constructor.
cl::size_t<3> region; region[0] = width; region[1] = height; region[2] = 1;
queue.enqueueWriteImage(src_image, CL_TRUE, origin, region, row_pitch, 0, ptr);
```

Example 5: Creation and initialization of luma source and reference image arguments

For native NV12 images, the channel orders and channel data types are restricted as (CL_NV12_INTEL, CL_UNORM_INT8). The attributes *CL_MEM_HOST_NO_ACCESS* and *CL_MEM_ACCESS_FLAGS_UNRESTRICTED_INTEL* must be set for NV12 images to indicate that the host may not directly read or write the NV12 image, and so that the Y and UV pane memory objects created in the host may be allowed to override the NV12 attributes to be able to read and write the individual planes.

For native NV12 images, it is possible to extract the luma and chroma planes using the cl_intel_planar_yuv extension API. In order to get accesses to the luma and chroma planes, two image objects need to be created as shown below with the *cl_mem* object set to the NV12 image object.

```
cl::ImageFormat imageFormatNV12(CL_NV12_INTEL, CL_UNORM_INT8);
cl::Image2D srcImage(
    context, CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
             CL_MEM_ACCESS_FLAGS_UNRESTRICTED_INTEL,
    imageFormatNV12, width, height);
cl::Image2D refImage(
    context, CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
             CL_MEM_ACCESS_FLAGS_UNRESTRICTED_INTEL,
    imageFormatNV12, width, height);

cl::Image2DYPlane srcYImage(context, CL_MEM_READ_ONLY, srcImage());
cl::Image2DYPlane refYImage(context, CL_MEM_READ_ONLY, refImage());

cl::Image2DUVPlane srcUVImage(context, CL_MEM_READ_ONLY, srcImage());
cl::Image2DUVPlane refUVImage(context, CL_MEM_READ_ONLY, refImage());
```

Example 6: Creation of NV12, luma and chroma plane image arguments.

The host program is responsible for populating the tiled image either using the clEnqueueWriteImage function or its wrapper function enqueueWriteImage, or using the clEnqueueMapImage or its wrapper function enqueueMapImage as shown below.

```
size_t pitchDestY = 0; cl_uchar * mappedAddrY = NULL;
size_t pitchDestUV = 0; cl_uchar * mappedAddrUV = NULL;
cl::size_t<3> origin, region;     // Init to 0 in constructor.

// Write luma.
region[0] = width; region[1] = height; region[2] = 1;
mappedAddrY =
    ( cl_uchar* )queue.enqueueMapImage(
        nv12ImageY, CL_TRUE, CL_MAP_WRITE, origin, region, &pitchDestY,
        NULL, NULL, NULL, &err );
for( cl_uint i = 0; i < height; ++i )
{
    memcpy(
        mappedAddrY + i * pitchDestY, yPlane + ( i * yPitch ),
        width );
}

err = queue.enqueueUnmapMemObject( nv12ImageY, mappedAddrY );
assert( err == CL_SUCCESS );

// Write chroma.
region[0] = width / 2; region[1] = height / 2; region[2] = 1;
mappedAddrUV =
    ( cl_uchar* )queue.enqueueMapImage(
        nv12ImageUV, CL_TRUE, CL_MAP_WRITE, origin, region, &pitchDestUV,
        NULL, NULL, NULL, &err );

cl_uchar * chromaDest = mappedAddrUV;
for( cl_uint j = 0; j < height / 2; j += 1 )
{
    cl_uchar * chromaTmp = chromaDest;
    for( cl_uint k = 0; k < width / 2; k += 1 )
    {
        chromaDest[0] = *( uPlaneSrc + ( j * uPitch ) + k );
        chromaDest[1] = *( vPlaneSrc + ( j * vPitch ) + k );
        chromaDest += 2;
    }
        chromaDest = chromaTmp + pitchDestUV;
}

err = queue.enqueueUnmapMemObject( nv12ImageUV, mappedAddrUV );
assert( err == CL_SUCCESS );
```

Example 7: Initialization of NV12, luma and chroma plane image arguments.

Source and Reference image parameters

The OpenCL compiler internally assigns a unique binding table index (BTI) for each kernel image parameter. The BTI indexes of consecutive kernel image parameters are assigned consecutively. The

VME unit accesses the source and reference images using its assigned BTIs. For source and reference images accessed by the VME unit there is an implicit assumption about the BTI of the source and reference images. The BTI of the first forward reference image parameter must immediately follow the BTI of the source image parameter, and the BTI of the first backward reference image parameter must immediately follow the BTI of the first forward reference image parameter. This implies that the source and reference image parameters must be specified in a certain order in a VME kernel as shown below; otherwise it will result in a complier reported error. Up to 16 pairs of forward and backward reference image parameters may be used in a VME kernel.

Note: OpenCL VME kernels may declare image parameters other than the source and reference images. These may be declared either before the source image or after the last reference image. The maximum number of image parameters supports can be queried using clGetDeviceInfo with CL_DEVICE_MAX_READ_IMAGE_ARGS and CL_DEVICE_MAX_WRITE_IMAGE_ARGS.

```
__kernel void  vme_custom(
    image2d_t src_image,
    image2d_t fwd_ref_image,
    image2d_t bwd_ref_image,
    ...
 )
```

Example 8: Source and reference parameter ordering requirements

Exclusive use of image parameters

The images processed by media blocks reads and writes require different surface state programming internally in the hardware as required for texture sampler reads. The state programming required for images processed by the VME unit is also different as required for images processed by the texture sampler or by media block read or write operations. The state programming is associated with the image binding table index (BTI) created by the OpenCL runtime uniquely for different kernel parameters. Therefore the same image kernel parameter cannot be used across these. In order to work-around this limitation, the user needs to use two separate image parameters in the OpenCL kernel, one for the VME operation and the other for the media block read/write operation. In the OCL host application, the same source image object can be bound to the two separate kernel arguments in order to read from the same image object in the kernel using VME and media block read operations.

```
__kernel void  vme_custom(
    image2d_t src_image,                // BTI 0
    image2d_t fwd_ref_image,            // BTI 1
    image2d_t bwd_ref_image,            // BTI 2
    image2d_t src_image_block_read,     // BTI 3
    ...
 )
{
    ...
    uint leftEdgeDW = intel_sub_group_media_block_read_ui(
        edgeCoord, 1, 16, src_image_block_read );
    ...
    result = intel_sub_group_avc_sic_evaluate_ipe(
        src_image, accelerator, payload );
}
```

Kernel code

```
...

cl::ImageFormat imageFormat(CL_R, CL_UNORM_INT8);
cl::Image2D srcImage(context, CL_MEM_READ_ONLY, imageFormat, width, height, 0,
0);
cl::Image2D ref0Image(context, CL_MEM_READ_ONLY, imageFormat, width, height, 0,
0);
cl::Image2D ref1Image(context, CL_MEM_READ_ONLY, imageFormat, width, height, 0,
0);
...
cl::Kernel kernel(p, "vme_custom");
kernel.setArg(0, srcImage);  // Same image used as VME source image
kernel.setArg(1, ref0Image);
kernel.setArg(2, ref1Image);
kernel.setArg(3, srcImage);   // Same image used as media block read image
...
```

Host code

Example 9: Exclusive use of image parameters by VME, media block read/write, and texture sampler

Inline media sampler

Inline media samplers are used to represent the VME accelerator in OpenCL kernels. There is no need to create host-side VME accelerator objects as was required with the previous host-side VME extensions. The VME built-in evaluation functions require a media sampler accelerator as one of its arguments. The inline media sampler only needs to be initialized as follows in the kernel.

```
...
sampler_t accelerator = CLK_AVC_ME_INITIALIZE_INTEL;
…
result = intel_sub_group_avc_sic_evaluate_ipe(
        src_image, accelerator, payload );
```

Example 10: Inline media sampler initialization

IME functions

IME is a fundamental and compute intensive part of VME. It is computed in the IME pipeline of the VME unit. It performs motion estimation at a full pixel resolution on a given source macroblock in a source image and a search window in a reference image to determine the best integer motion vectors, their associated distortions, and the best macroblock shape decision combination.

The IME operation can be broken down into two stages as shown in the figure below.

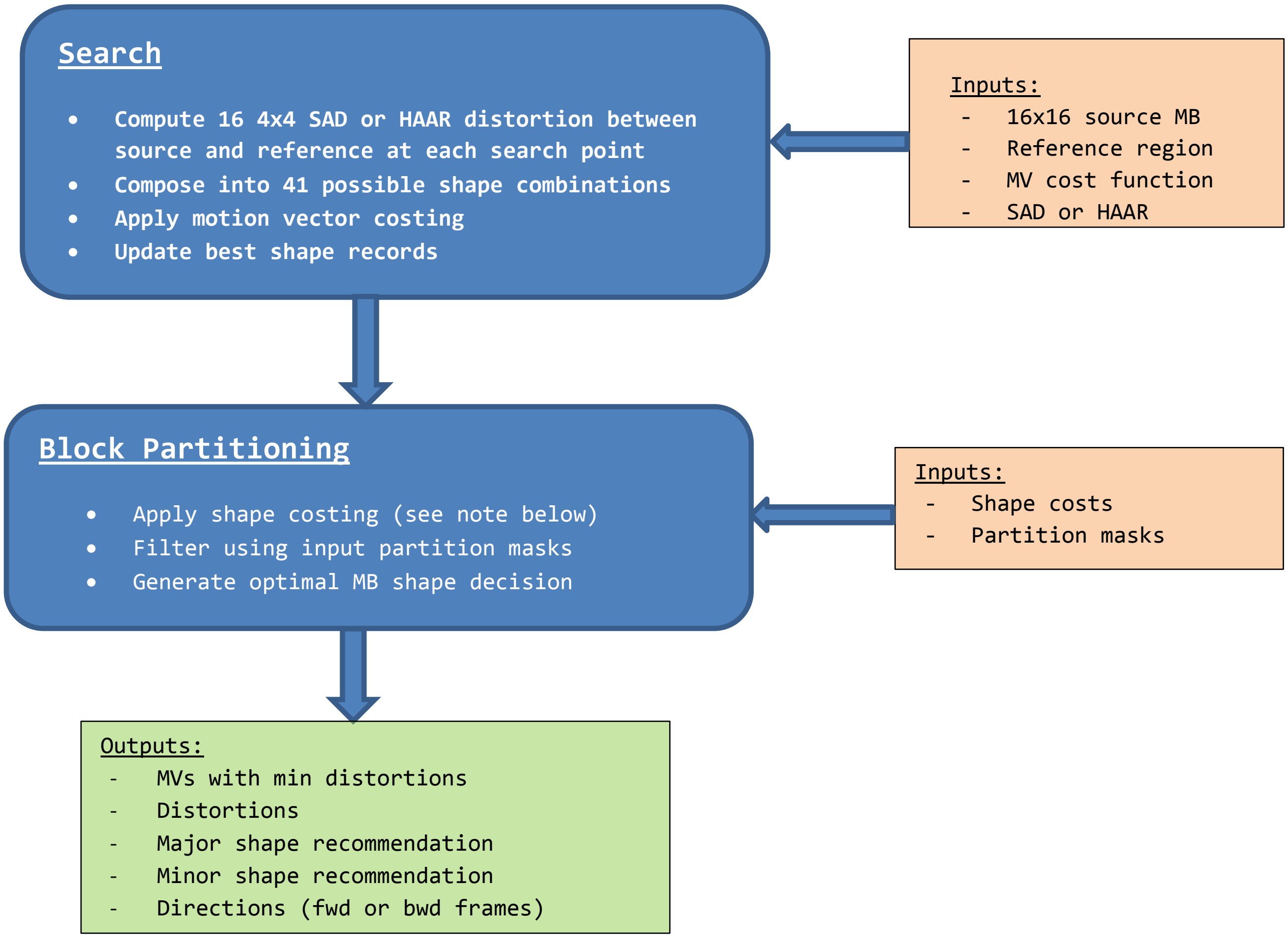

Search
Compute 16 4x4 SAD or HAAR distortion between source and reference at each search point
Compose into 41 possible shape combinations
Apply motion vector costing
Update best shape records
Inputs:
- 16x16 source MB
- Reference region
- MV cost function
- SAD or HAAR
Block Partitioning
Apply shape costing (see note below)
Filter using input partition masks
Generate optimal MB shape decision
Inputs:
- Shape costs
- Partition masks
Outputs:
- MVs with min distortions
- Distortions
- Major shape recommendation
- Minor shape recommendation
- Directions (fwd or bwd frames)

Figure 7: IME operation

Note: Penalties can be applied to the SAD or HAAR values to bias against major or minor shapes (shape penalty) based on the targeted QP and slice type (I, P, B). The inputs for IME evaluation will include the shape costs for each shape and should be set based on the image analysis.

A flowchart of the IME function phases to evaluate an IME operation is shown in Figure 8.

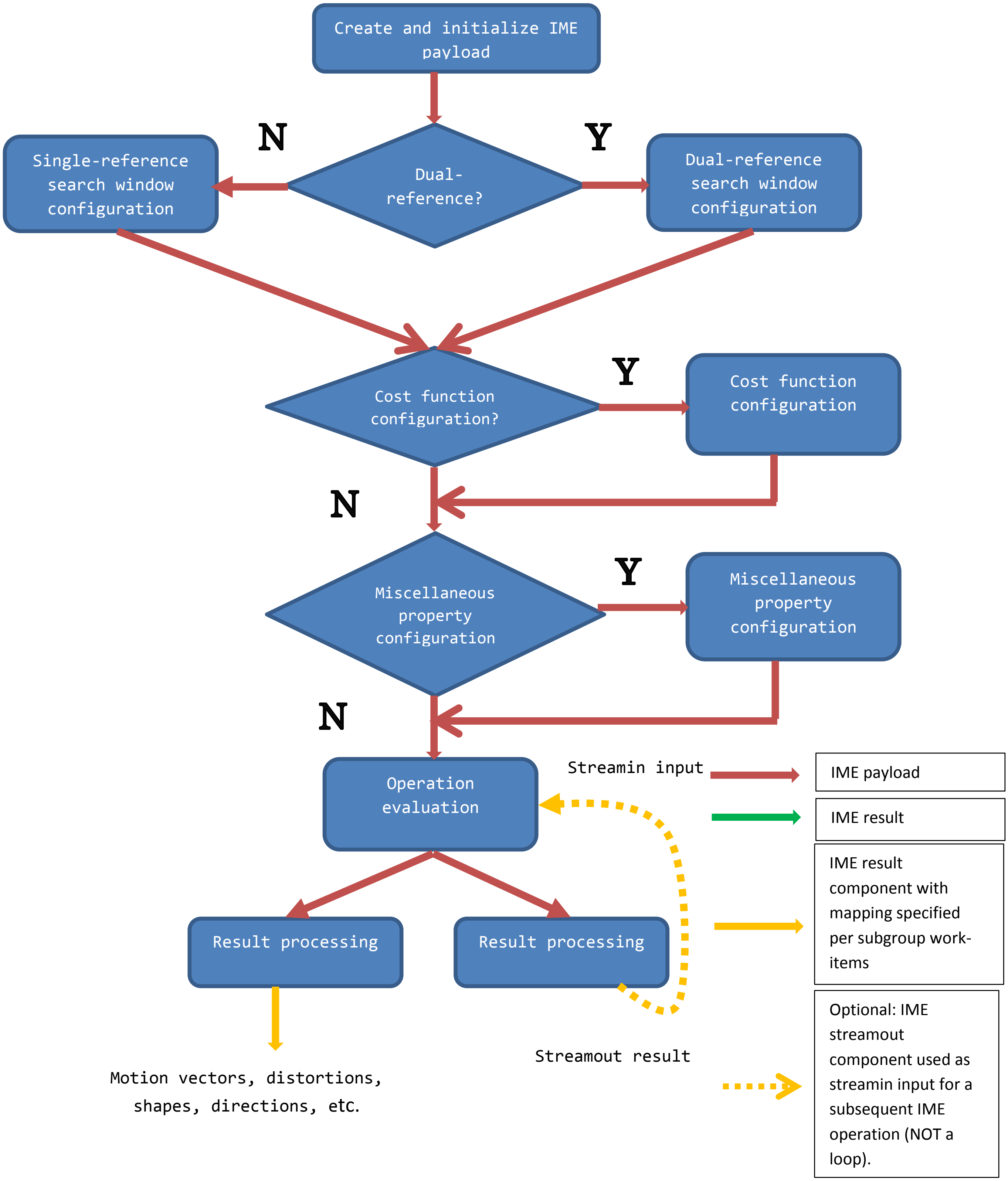

Create and initialize IME payload
N
Y
Single-reference search window configuration
Dual-reference?
Dual-reference search window configuration
Y
Cost function configuration?
Cost function configuration
N
Y
Miscellaneous property configuration
Miscellaneous property configuration
N
Operation evaluation
Streamin input
IME payload
IME result
Result processing
Result processing
IME result component with mapping specified per subgroup work-items
Motion vectors, distortions, shapes, directions, etc.
Streamout result
Optional: IME streamout component used as streamin input for a subsequent IME operation (NOT a loop).

Figure 8: IME function phases flow chart

The following section gives brief descriptions of the various function phases built-in functions and shows examples of their uses. Refer to the [4] for detailed function descriptions and their restrictions.

Initialization phase

intel_sub_group_avc_ime_payload_t
intel_sub_group_avc_ime_initialize(
  ushort2 src_coord,
  uchar partition_mask,
  uchar sad_adjustment)

Description

Return an initialized payload for an IME message in a SIMD16 kernel.

The coordinates of the source MB in pixel units relative to the top-left corner of the source image, the allowed partition mask, and SAD or HAAR transform setting for the per pixel differences between the source macroblock and reference region is set in the initialized payload. If the source image is an interlaced image, then see section [Interlaced content processing] for additional information on the specification of source coordinates.

Example

```
ushort2  src_coord;
  ...
uchar  partition_mask =
  CLK_AVC_ME_PARTITION_MASK_16x16_INTEL &
  CLK_AVC_ME_PARTITION_MASK_8x8_INTEL &
  CLK_AVC_ME_PARTITION_MASK_16x8_INTEL &
  CLK_AVC_ME_PARTITION_MASK_8x16_INTEL;

intel_sub_group_avc_ime_payload_t payload =
  intel_sub_group_avc_ime_initialize(
    src_coord,                             // source MB offset in pixel units
    partition_mask,                        // enable all major shapes
    CLK_AVC_ME_SAD_ADJUST_MODE_HAAR_INTEL  // HAAR transformed distortions
  );
```

Example 11: IME initialization

Reference window configuration phase

intel_sub_group_avc_ime_payload_t
intel_sub_group_avc_ime_set_single_reference(
  short2 ref_offset,
  uchar search_window_config,
  intel_sub_group_avc_ime_payload_t payload)

Description

Updates the input payload for a single-reference search with the configurations for the reference search regions, and return the updated payload.

The configuration of the search region specifies its dimensions and its location within the reference image. The default (with ref_offset set to zero) reference region is around the co-located MB in the reference image. The reference region can be offset by a 2D integer offset (ref_offset) as shown in the figure below. If the reference image is an interlaced image, then see section [Interlaced content processing] for additional information on the specification of the reference region offset (ref_offset).

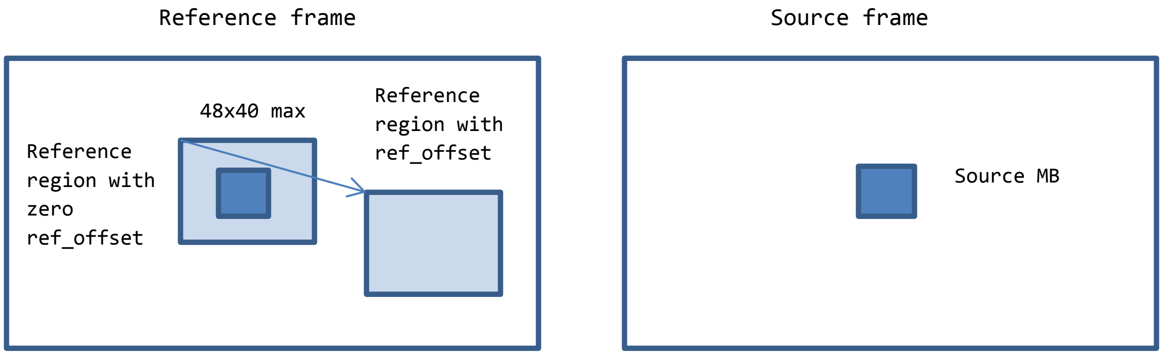

Reference frame
Source frame
48x40 max
Reference region with ref_offset
Reference region with zero ref_offset
Source MB

Figure 9: Single reference region offsetting

The VME unit has a 2K reference cache which is used for loading the reference search regions for IME search.

Possible reference region search locations are grouped in a predefined 4x4 pattern, and all locations within the same group must be completely chosen or completely skipped. These predefined groups are called *search units*.

The path taken during searching in a reference region is referred to as a *search path*. The steps taken in a search path are in units of search units. The search path must lie within the defined search region; all search units that are outside of the search region are skipped.

There are essentially two kinds of searches. The first is an *exhaustive* search in which all the search units per a pre-defined search path are exhaustively searched in a spiral pattern with the search center being in the middle of the search region. The second is a *diamond* search in which the initial part of the search path is in a diamond pattern as per a pre-defined search path, and for the latter part of the search path gradient based searching is used based on the best results from the initial pre-defined part of the search for up to a maximum of 57 search units.

The pre-determined search paths for an exhaustive and a diamond search are illustrated in the figures below.

SU (X,Y)	0	1	2	3	4	5	6	7
0	47	30	31	32	33	34	35	36
1	46	29	12	13	14	15	16	37
2	45	28	11	2	3	4	17	38
3	44	27	10	1	0	5	18	39
4	43	26	9	8	7	6	19	40
5	42	25	24	23	22	21	20	41

Figure 10: Exhaustive search path for a 48x40 search window

SU (X,Y)	0	1	2	3	4	5	6	7
0				27	15			
1			26	17	6	19		
2		25	16	3	2	11	20	
3	24	13	5	1	0	4	7	14
4		28	18	10	8	12	22	31
5			29	21	9	23	30	

Figure 11: Diamond search path (pre-determined part) for a 48x40 search window

Either one or two reference search regions can be loaded into the reference cache for single reference and dual-reference searches respectively. The predefined single reference search regions defined based on these hardware restrictions are as follow.

EXHAUSTIVE	48x40 search region with exhaustive single reference search; the search path length is 48
SMALL	28x28 search region with exhaustive search; the search path length is 9
TINY	24x24 search region with exhaustive search; the search path length is 4
EXTRA TINY	20x20 search region with exhaustive search; the search path length is 1
DIAMOND	48x40 search region with diamond single reference search; the initial pre-defined search path length is 16; the total search path length is 57
LARGE DIAMOND	48x40 search region with large diamond single reference search; the initial pre-defined search path length is 32; the total search path length is 57

Table 2: Single-reference search window configurations

Since we search 2x the number of SUs in LARGE DIAMOND w.r.t DIAMOND and then continue with adaptive based on the best results with fixed search, the quality will be better with LARGE DIAMOND but the performance will be lesser. For 48x40 search windows, the EXHAUSTIVE configuration is commonly used and produces the best quality results, but the DIAMOND configuration is almost twice as fast, with the LARGE_DIAMOND configuration being a trade-off between the two in terms of quality and performance.

Example

```
// Configure the initialized IME payload for a single reference search.
// The search window is 48x40 with exhaustive search, with its location
// offset specified by the ref_offset.

intel_sub_group_avc_ime_payload_t payload =
  intel_sub_group_avc_ime_set_single_reference(
    ref_offset,                               // reference window offset
    CLK_AVC_ME_SEARCH_WINDOW_EXHAUSTIVE_INTEL,// 48x40 reference window size
    payload                                   // previous initialized payload
  );
```

Example 12: IME single reference search configuration

intel_sub_group_avc_ime_payload_t
intel_sub_group_avc_ime_set_dual_reference(
   short2 fwd_ref_offset,
   short2 bwd_ref_offset,
   uchar search_window_config,
   intel_sub_group_avc_ime_payload_t payload)

Description

Updates the input payload for a dual-reference search with the configurations for the search regions in both the forward (L0) and backward (L1) reference images, and returns the updated payload.

The search regions in both the forward and backward reference images will be configured with the same dimensions as specified by the *search_window_config* parameter, but they can be independently located using the *fwd_ref_offset* and bwd*_ref_offset* parameters. If a reference image is an interlaced image, then see section [Interlaced content processing] for additional information on the specification of the reference region offset (*fwd_ref_offset* or *bwd_ref_offset*).

The pre-determined search paths for an exhaustive and a diamond search for a single reference are illustrated in the figures below.

SU (X,Y)	0	1	2	3
0	12	13	14	15
1	11	2	3	4
2	10	1	0	5
3	9	8	7	6

Figure 12: Exhaustive search path for a 32x32 search window

SU (X,Y)	0	1	2	3
0			6	
1		3	2	9
2	5	1	0	4
3		8	7	

Figure 13: Diamond search path (pre-determined part) for a 32x32 search window

Based on the 2K reference cache size in VME which is used for loading the dual reference search regions for dual-reference IME search, the following pre-defined search windows are defined.

EXHAUSTIVE	32x32 dual search region for exhaustive dual-reference search; the search path length is 16 per reference
SMALL	28x28 search region with exhaustive search; the search path length is 9 per reference
TINY	24x24 search region with exhaustive search; the search path length is 4 per reference
EXTRA TINY	20x20 search region with exhaustive search; the search path length is 1 per reference
DIAMOND	32x32 dual search region for diamond dual-reference search; the initial pre-defined search path length is 7 per reference ; the total search path length is 57 per reference
LARGE DIAMOND	32x32 dual search region for large diamond dual-reference search; the initial pre-defined search path length is 10 per reference; the total search path length is 57 per reference

Table 3: Dual-reference search window configurations

Identical search window sizes and search paths are used to search both the forward and backward reference images; only their search window offsets may be independently specified.

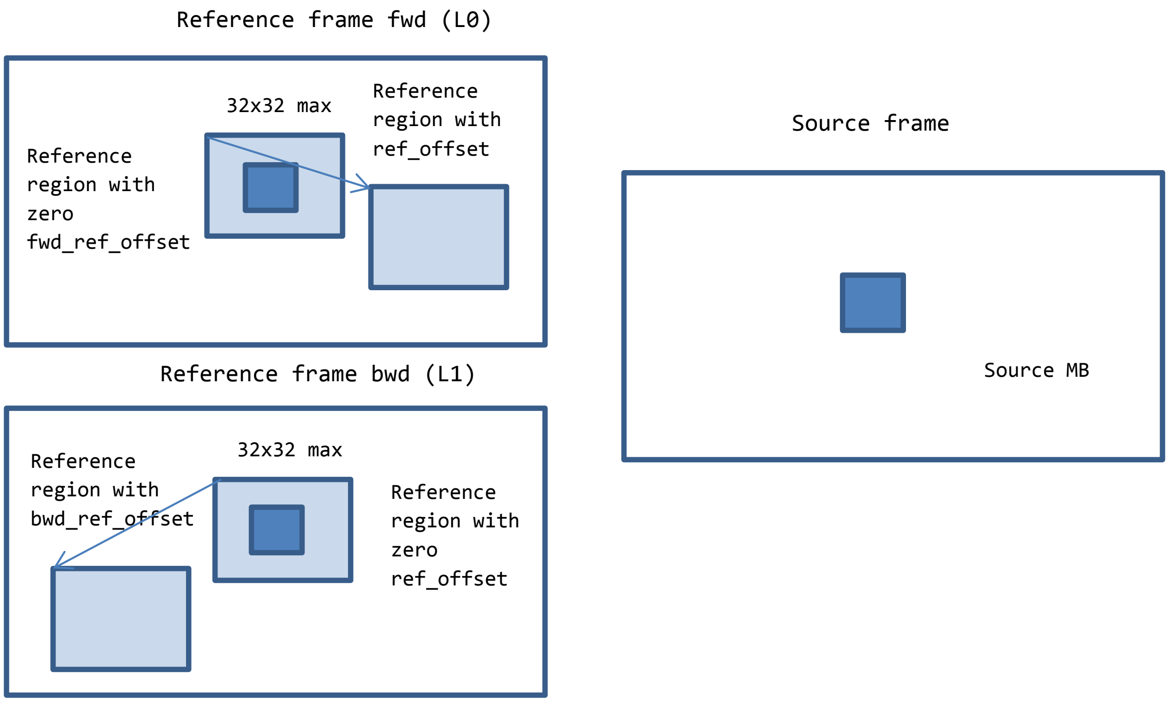

Reference frame fwd (L0)
32x32 max
Reference region with ref_offset
Reference region with zero fwd_ref_offset
Source frame
Source MB
Reference frame bwd (L1)
Reference region with bwd_ref_offset
32x32 max
Reference region with zero ref_offset

Figure 14: Dual-reference search region offsetting

Example

```
...

// Configure the initialized IME payload for a dual reference search.
// The search windows are 32x32 with exhaustive search, with the forward
// location offset specified by fwd_ref_offset and backward ref offset
// specified by bwd_ref_offset.

intel_sub_group_avc_ime_payload_t payload =
  intel_sub_group_avc_ime_set_dual_reference(
    fwd_ref_offset,                          // fwd reference window offset
    bwd_ref_offset,                          // bwd reference window offset
    CLK_AVC_ME_SEARCH_WINDOW_EXHAUSTIVE_INTEL,// 32x32 reference window size
    payload                                   // previous initialized payload
  );
```

Example 13: IME dual-reference search configuration

ushort2
intel_sub_group_avc_ime_adjust_ref_offset *(*
  short2 ref_offset,
  ushort2 src_coord,
  ushort2 ref_window_size,
  ushort2 image_size *)*

Description

The reference offset argument for the IME configuration functions *intel_sub_group_avc_ime_set_dual_reference* or *intel_sub_group_avc_ime_set_dual_reference* needs to be such that the reference search region is at least partially within the reference image, otherwise the IME evaluation results are undefined. This function will return an adjusted reference offset such that the reference search region is on the appropriate image boundary for the given source coordinate if the input reference offset caused the reference search region to be fully out-of-bounds, otherwise it simply returns the input reference offset.

If the reference image is an interlaced image, then see section [Interlaced content processing] for additional information on the specification of the reference region offset (ref_offset). Since, the actual layout of the top and bottom fields in the reference image is in an interleaved fashion, the height of the top or bottom fields should be exactly half of the actual reference image height.

Example

```
...
// Get the reference search window 2D size for a single reference search.
ushort2 ref_window_size =
  intel_sub_group_ime_ref_window_size(
    CLK_AVC_ME_SEARCH_WINDOW_EXHAUSTIVE_INTEL,
    false );

// Adjust IME reference window offsets so that the reference windows stay
// within the reference frame
ref_offset =
  intel_sub_group_avc_ime_adjust_ref_offset(
    ref_offset,
    src_coord,
    ref_window_size,
    convert_ushort2( get_image_dim( ref_image ) ) );

intel_sub_group_avc_ime_payload_t payload =
  intel_sub_group_avc_ime_set_single_reference(
    ref_offset,                               // reference window offset
    CLK_AVC_ME_SEARCH_WINDOW_EXHAUSTIVE_INTEL,// 48x40 reference window size
    payload                                   // previous initialized payload
  );
```

Example 14: Ensuring in-bounds IME reference search window

Cost configuration phase

The cost functions are a set of functions used to configure the heuristics designed in the VME unit to model the rate-distortion optimization (RDO) with the objective to minimize the bits in encoding. Ultimately the distortion is the measure used for the rate-distortion value and these cost configuration function alter the computed distortion values. The manner in which the VME unit calculates the distortion for inter estimation is shown in [Figure 15: Inter distortion calculation].

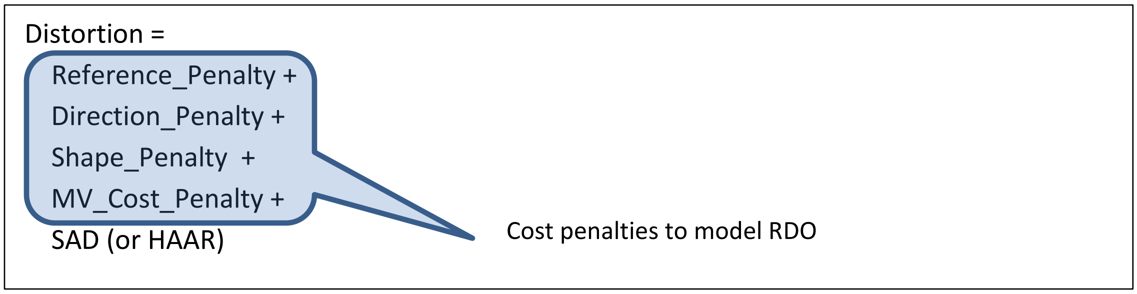


Figure 15: Inter distortion calculation

Good default values to use are provided by the API, by means of built-in functions, to configure the various cost penalties. These default values are derived based on analysis done across a wide range of video content, and can be obtained per slice type and the quantization parameter (Qp) value. The API, however, does allow for the provision to override the default values.

> **Note:** It must be noted that "default" cost configuration does not mean a "zero" cost configuration. During payload initialization, payloads are initialized with zero cost configuration values. When the payload is evaluated with a zero cost configuration, the distortion is simply equal to the pure SAD values. On the other hand, if any of the cost configuration functions have been used to configure the payload, then the returned distortions no longer are equal to the pure SAD values.

The cost configuration functions are categorized as follows.

Inter cost configurations

Multi-reference cost configuration

```
uchar
intel_sub_group_avc_mce_get_default_inter_base_multi_reference_penalty(
   uchar slice_type,
   uchar qp  )

intel_sub_group_avc_ime_payload_t
intel_sub_group_avc_ime_set_inter_base_multi_reference_penalty(
   uchar reference_base_penalty,
   intel_sub_group_avc_ime_payload_t payload )
```

Description

This function updates the input payload to specify a reference base penalty when HW assisted multi-reference search is performed. The objective of this cost penalty is to minimize the bits in encoding the header by biasing major block partitions from nearer reference images. Each reference major partition block gets associated with a penalty based on the distance of the reference image of the block from the source image. The distance of the reference images is determined as described in section [Source and reference]. The base penalty is scaled using a scaling factor based on the implied distance of the reference image from the source image as illustrated below.

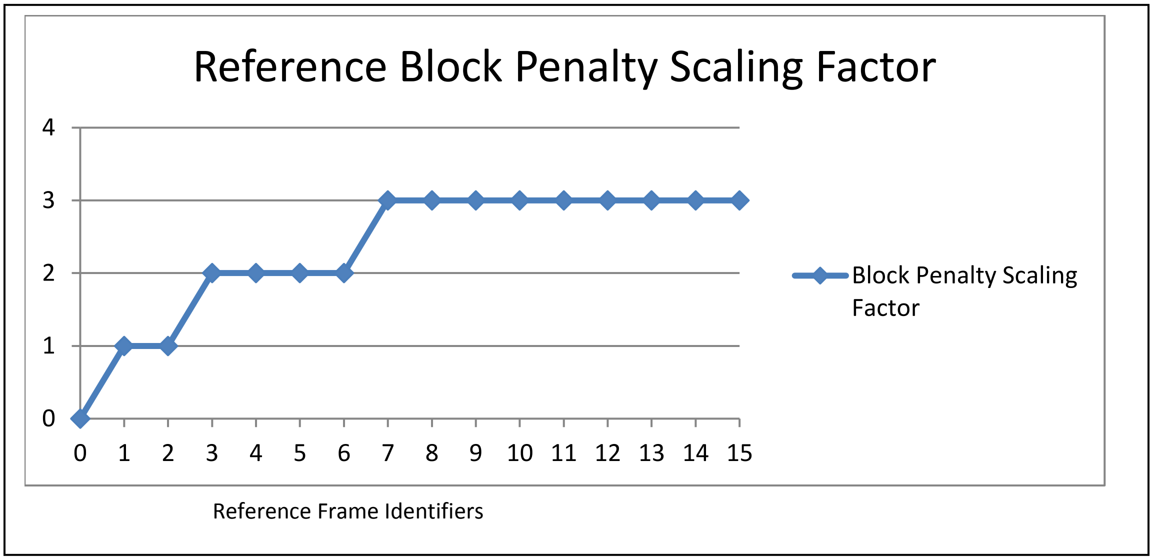


Figure 16: Multi-reference Scaling Factor By Reference Identifier

The value of base penalty is specified in U4U4 format and the integer value must fit within 12 bits.

Example

```
...
// Get a good default base penalty to use.
uchar ref_penalty =
  intel_sub_group_avc_mce_get_default_inter_base_multi_reference_penalty(
      slice_type, qp );
// Update the payload with the cost penalty for multi-reference search.
payload =
  intel_sub_group_avc_ime_set_ set_inter_base_multi_reference_penalty(
      ref_penalty, payload );
```

Example 15: Multi-reference cost penalty configuration

Direction cost configuration

uchar

intel_sub_group_avc_mce_get_default_inter_direction_penalty(

uchar slice_type,
uchar qp *)*

intel_sub_group_avc_ime_payload_t
intel_sub_group_avc_ime_set_inter_direction_penalty(
uchar direction_cost,
intel_sub_group_avc_ime_payload_t payload)

Description

This function updates the input payload to specify a direction penalty for backward reference images. A flat penalty is added for all blocks from backward reference images. The value of the penalty must be in U4U4 format and its decoded integer value must bit fit in 12 bits.

Example

```
...
// Get a good default direction penalty to use.
uchar dir_penalty =
  intel_sub_group_avc_ime_get_default_inter_direction_penalty(
      slice_type, qp );
// Update the payload with the direction cost penalty.
payload =
  intel_sub_group_avc_ime_set_inter_direction_penalty(
      dir_penalty, payload );
```

Example 16: Direction cost penalty configuration

Inter shape cost configuration

ulong
intel_sub_group_avc_mce_get_default_inter_shape_penalty(
uchar slice_type,
uchar qp *)*

intel_sub_group_avc_mce_payload_t
intel_sub_group_avc_mce_set_inter_shape_penalty(
ulong packed_shape_penalty,
intel_sub_group_avc_mce_payload_t payload)

Description

This function updates the input payload to specify the shape penalty for inter motion estimation. The shape penalty is specified as an unsigned long integer value with the bits specifying the shape penalty in U4U4 format as shown in the table below.

Bits	Penalty for Shape
7:0	16x8 and 8x16
15:8	8x8
23:16	8x4 and 4x8
31:24	4x4
39:32	16x16
63:40	Must be zero

Table 4: Inter packed shape penalty format

The U4U4 decoded integer values for byte 0 and byte 4 must bit fit in 12 bits, while the U4U4 decoded integer values for the other bytes must fit within 10 bits.

Example

```
...
// Get a good default shape penalties to use.
ulong shape_penalties =
  intel_sub_group_avc_mce_get_default_inter_shape_penalty(
      slice_type, qp );
// Update the payload with the shape cost penalties.
payload =
  intel_sub_group_avc_mce_set_inter_shape_penalty(
      shape_penalties, payload );
```

Example 17: Inter shape cost penalty configuration

The API provides good default values to use for shape costs, but provisions exist for providing custom values. A rough model for specifying custom values is provided below.

```
lambda = pow(2, (qp-12)/6);
bias = 2.0;
union {
   ulong      raw;
   struct {
      uchar s0;           // 16x8, 8x16
      uchar s1;           // 8x8
      uchar s2;           // 8x4, 4x8
      uchar s3;           // 4x4
      uchar s4;           // 16x16
      uchar s5, s6, s7;   // Must be zero.
   } bits;
} packedShapeCost;

packedShapeCost.raw = 0;

float InterShapeCost[5] =
{
                // PSLICE
    4.0,        // 16x8, 8x16
    1.0,        // 8x8q
    2.0,        // 8x4q, 4x8q, 16x8_FIELD
    3.0,        // 4x4q, 8x8_FIELD
    3.0,        // 16x16
};

packedShapeCost.bits.s0 = min( ConvertToU4U4((U16)( lambda *
   InterShapeCost[ INTER_16x8 ]   * bias )), 0x8f );
packedShapeCost.bits.s1 = min( ConvertToU4U4((U16)( lambda *
   InterShapeCost[ INTER_8x8 ]  * bias )), 0x6f );
packedShapeCost.bits.s2 = min( ConvertToU4U4((U16)( lambda *
   InterShapeCost[ INTER_8x4 ]  * bias )), 0x6f );
packedShapeCost.bits.s3 = min( ConvertToU4U4((U16)( lambda *
   InterShapeCost[ INTER_4x4 ]  * bias )), 0x6f );
packedShapeCost.bits.s4 = min( ConvertToU4U4((U16)( lambda *
   InterShapeCost[ INTER_16x16 ] * bias )), 0x8f );
payload_t =  intel_sub_group_avc_mce_set_inter_shape_penalty(
   packedShapeCost.raw, payload )
```

Figure 17: Sample model for custom inter shape costs

Motion vector cost configuration

intel_sub_group_avc_ime_payload_t
intel_sub_group_avc_ime_set_motion_vector_cost_function(
   ulong packed_cost_center_delta,
   uint2 packed_cost_table,
   uchar cost_precision,
   intel_sub_group_avc_ime_payload_t payload)

Description

This function updates the input payload to specify a piecewise linear cost function for motion vectors and returns it. The objective of the cost function is to minimize the bits in differentially encoding the motion vectors by biasing the returned best motion vectors closer to their cost centers. The cost center is typically decided to be the predicted motion vector from the neighboring macroblocks. Cost centers are specified independently in QPEL resolution as offsets from the top left corner of the source macroblock for each of the four 8x8 block partitions of a macroblock, and for both forward and backward reference regions. For 16x16 partitions work-item '0' provides the cost center. For 8x16 partitions work-items '0' and '1' provide the cost centers. For 16x8 partitions work-items '0' and '2' provide the cost centers.

The packed cost table specifies the cost penalties for 8 control points in U4U4 format. Given a motion vector for a shape, mv, the motion vector delta, dx, is defined to be its difference from the given cost center, cc, i.e. dx = |mv –cc|. The costing penalty is based on dx. The cost penalty is a piecewise linear interpolation from the cost penalty table, LUT_MV, which is defined based on powers-of-2 integer motion vector deltas. The piecewise linear interpolation is performed using QPEL precision. The maximum distance provided in the table is 64 pixels when the cost precision is in PEL units. A linear ramp with gradient of 1 on integer distance is applied for bigger distances with maximum penalty capped to 0xFF. The range of the cost function can be adjusted based on the cost precision settings for the motion vector deltas as follows:

Cost Precision	Motion Vector Delta Unit	Cost range in Pixels
CLK_AVC_ME_COST_PRECISION_PEL_INTEL	pixel	0-63
CLK_AVC_ME_COST_PRECISION_DPEL_INTEL	dual pixel	0-127
CLK_AVC_ME_COST_PRECISION_HPEL_INTEL	half pixel	0-31
CLK_AVC_ME_COST_PRECISION_QPEL_INTEL	quarter pixel	0-15

Table 5: Inter motion vector cost precision units and ranges

The exact cost function calculation is as follows, where the unit of dx is as per the cost precision specified.

```
// Special handling for small deltas.
Costing_penalty_x = LUT_MV[int(DX)], if dx < 3 and dx = int(dx);

// Lookup for powers-of-2 deltas.
Costing_penalty_x = LUT_MV[p+1], else if dx = 2^p , for any p ≤ 6;.

// Interpolation for non-powers of 2 deltas.
Costing_penalty_x = LUT_MV[p+1] + ((LUT_MV[p+2] – LUT_MV[p+1])*k)>>p,
else if dx = 2^p + k, for any p< 6 and k< 2^p , and

// Special handling for out of normal cost function range.
Costing_penalty_x = min (LUT_MV[7] + int(dx) – 64, 255), else if dx > 64.

The total costing penalty for a motion vector is
Costing_penalty = Costing_penalty_x + Costing_penalty_y
```

Figure 18: Cost function description

The cost penalty table values are specified in the U4U4 format, which represents a value of (B<<S), where B, called base, is the 4 bit LSB of the byte and S, called shift, is the 4 bit MSB of the byte. The values in the cost penalty table are typically based on the Qp and can be provided using the argument ‘packed_cost_table’.

Example

```
  ...
  // Using the reference offset as the cost center after converting to
  // QPEL resolution.
  cost_coord.s0 = ref_offset.s0 << 2;
  cost_coord.s1 = ref_offset.s1 << 2;

  // Set the unidirectional cost center in the expected format.
  // Here all 4 cost-centers for all 8x8 partitions in work-items 0 to 3 are
  // set to the same value. They can potentially be set to different values.
  uint2 packed_cost_center_delta_int;
  packed_cost_center_delta_int.s0 = as_uint( cost_coord );
  packed_cost_center_delta_int.s1 = 0;
  ulong packed_cost_center_delta = as_ulong( packed_cost_center_delta_int );

  // Update the payload with the cost function.
  payload =
    intel_sub_group_avc_ime_set_motion_vector_cost_function(
        packed_cost_center_delta,
        intel_sub_group_avc_mce_get_default_low_penalty_cost_table(),
        CLK_AVC_ME_COST_PRECISION_QPEL_INTEL,
        payload );
```

Example 18: Motion vector cost function configuration

The API provides good default values to use for the cost table, but provisions exist for providing custom values. A rough model for specifying custom values is provided below.

```
lambda = pow(2, (qp-12)/6);      // from AVC reference encoder
bias = 2.0;
union {
   uint2  raw;
   uchar  points[8];
} packedCostTable;

packedCostTable.raw = 0;

double InterMVCost[8] = { 1.0, 2.0, 3.0, 5.0, 5.0, 6.0, 7.0, 8.0 };

for( int i = 0; i < 8; i++ ) {
   packedCostTable.bits.points [ i ]= min( ConvertToU4U4((U16)( lambda *
      InterMVCost[ i ] * bias )), 0x6f );
}
...
payload =
  intel_sub_group_avc_ime_set_motion_vector_cost_function(
      packed_cost_center_delta,
      packedCostTable.raw,
      CLK_AVC_ME_COST_PRECISION_QPEL_INTEL,
      payload );
```

Figure 19: Sample model for custom motion vector cost table

Miscellaneous property configuration phase

Interlaced content processing

The VME unit can natively process interlaced images, processing the top or bottom field MB from a MB pair in a source or reference image. However two separate VME (IME, REF, or SIC) evaluation calls are required for evaluating the top and bottom field MBs, one for each.

In theory, it is equivalent to creating two separate half height images, one with the top field lines and another with the bottom field lines, and then performing VME evaluation operations on both the top and bottom field images.

It is to be noted that for the purposes of specifying the MB and reference region coordinates for VME operations, the top MBs are considered as logically overlapping with the bottom MBs (i.e. the bottom field lines are considered as logically overlapping with the top field lines).

intel_sub_group_avc_mce_payload_t
intel_sub_group_avc_ime_set_source_interlaced_field_polarity(
   uchar *src*_field_polarity,
   intel_sub_group_avc_mce_payload_t payload)

intel_sub_group_avc_mce_payload_t
intel_sub_group_avc_ime_set_single_reference_interlaced_field_polarity(
   uchar ref_field_polarity,

intel_sub_group_avc_mce_payload_t payload)

intel_sub_group_avc_mce_payload_t
intel_sub_group_avc_ime_set_dual_reference_interlaced_field_polarities(
uchar *fwd_*ref_field_polarity,
uchar *bwd_*ref_field_polarity,
intel_sub_group_avc_mce_payload_t payload)

Description

These functions are used to specify that the source and reference images are interlaced and the field lines to process (field polarities).

Example

```
...
intel_sub_group_avc_ime_payload_t payload =
   intel_sub_group_avc_ime_initialize(srcCoord, partition_mask, sad_adjustment);
payload =
   intel_sub_group_avc_ime_set_single_reference(
       refCoord, CLK_AVC_ME_SEARCH_WINDOW_EXHAUSTIVE_INTEL, payload);
payload =
   intel_sub_group_avc_ime_set_motion_vector_cost_function(
       cost_center, packed_cost_table, search_cost_precision, payload );
payload =
   intel_sub_group_avc_ime_set_source_interlaced_field_polarity(
       polarity, payload );
payload =
   intel_sub_group_avc_ime_set_single_reference_interlaced_field_polarity(
       polarity, payload);
intel_sub_group_avc_ime_result_t result =
   intel_sub_group_avc_ime_evaluate_with_single_reference(
       srcImg, refImg, vme_sampler, payload )
```

Example 19: Configuration of source and reference field polarities

Evaluation and result processing phase

intel_sub_group_avc_ime_result_t
intel_sub_group_avc_ime_evaluate_with_single_reference(
image2d_t src_image,
image2d_t ref_image,
sampler_t vme_accelerator,
intel_sub_group_avc_ime_payload_t payload)

intel_sub_group_avc_ime_result_t

intel_sub_group_avc_ime_evaluate_with_dual_reference(
    image2d_t src_image*,*
    *image2d_t fwd*_ref_image*,*
    *image2d_t bwd*_ref_image*,*
    sampler_t vme_accelerator,
    intel_sub_group_avc_ime_payload_t payload)

Description

These functions perform the actual IME operation in the IME pipeline of the VME unit based on the configured payload, and where the operation latency is incurred. One version of the function is for single reference evaluations, while the other is for dual-reference evaluations. If the dual-reference version is used, the best motion vectors are returned for both the forward and backward reference images. The distortions are returned only for the best of forward or backward reference images, along with the best directions for major partitions. The result is returned in an object of an opaque type intel_sub_group_avc_ime_result_t. The result processing phase functions are required to extract the various results components.

uchar
intel_sub_group_avc_ime_get_inter_major_shape(
    intel_sub_group_avc_ime_result_t result *)*

uchar
intel_sub_group_avc_ime_get_inter_minor_shapes(
    intel_sub_group_avc_ime_result_t result *)*

Description

The above two functions return the block partitioning recommendation from the VME unit. Refer to [4] for a precise description of the interpretation of the result values.

uchar
intel_sub_group_avc_ime_get_inter_directions(
    intel_sub_group_avc_ime_result_t result *)*

Description

If the IME operation was configured for a dual-reference search, this function indicates whether the better match was found for the forward or backward reference image for each major partition. Refer to [4] for a precise description of the interpretation of the result values.

ulong

intel_sub_group_avc_ime_get_motion_vectors(
intel_sub_group_avc_ime_result_t result)

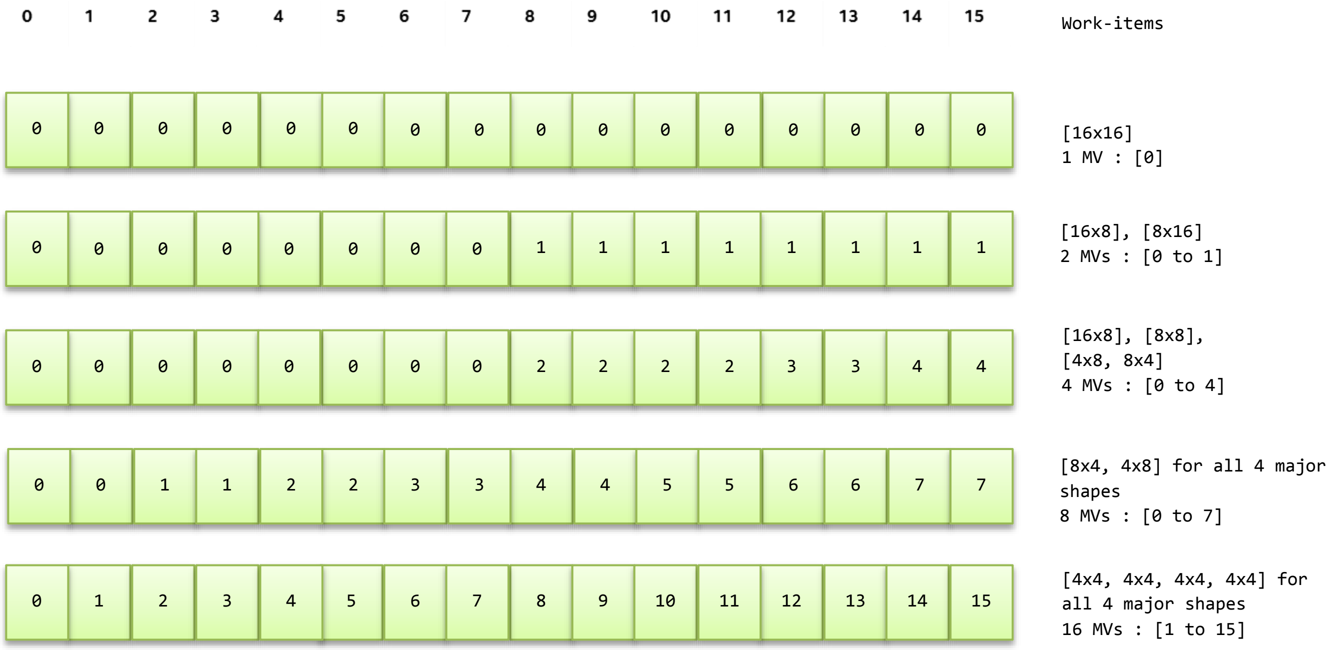
Description

Up to 16 packed bidirectional MVs are returned, one per work-item. The MVs are returned in QPEL resolution. If the search operation's payload was setup for unidirectional search then only the forward packed MV will be valid in each packed bidirectional MV, otherwise both packed MVs will be valid. The MVs returned are those with the minimum distortion within the forward (and backward for dual-reference) reference search region. The distortion is computed as shown in [Figure 15: Inter distortion calculation] to approximately take in to account the bits to encode the SAD, motion vector and the shape w.r.t to the neighboring macroblocks.

The packed bidirectional MVs have to be selected by their respective work-items based on the result block major and minor shapes as shown in the figure below.

Note: All sub-block bidirectional MVs per get replicated for each partition. For example, for a 16x16 partition, all smaller sub-block bidirectional MVs are replicated to the same bidirectional MV, and for 8x8 partitions, each 8x8 must have its respective sub-block bidirectional MVs replicated. This is not important to extract the component bidirectional MVs itself, but is needed if the result of this function is used to initialize the input motion vectors to a REF initialization function.

Note: With interlaced images, the MBs for the top field MBs are considered as logically overlapping with the bottom MBs.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Work-items
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	[16x16] 1 MV : [0]
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	[16x8], [8x16] 2 MVs : [0 to 1]
0	0	0	0	0	0	0	0	2	2	2	2	3	3	4	4	[16x8], [8x8], [4x8, 8x4] 4 MVs : [0 to 4]
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	[8x4, 4x8] for all 4 major shapes 8 MVs : [0 to 7]
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	[4x4, 4x4, 4x4, 4x4] for all 4 major shapes 16 MVs : [1 to 15]

Figure 20: IME motion-vector work-item mapping

ushort
intel_sub_group_avc_ime_get_inter_distortions(
  intel_sub_group_avc_ime_result_t result *)*

Description

This function returns the inter distortions result corresponding to the MVs returned by *intel_sub_group_avc_ime_get_motion_vectors(..)*. The distortion computation is as shown in [Figure 15: Inter distortion calculation]. The inter directions result returned by *intel_sub_group_avc_ime_get_inter_directions(..)* will specify if the distortion corresponds to the forward MV or backward MV. Up to 16 distortions are returned, one per work-item. The distortions have to be selected by their respective work-items based on the result block major and minor shapes just as for the result MVs as shown above in Figure 20.

Example

```
...
// Evaluate the IME operation with its configured payload.
// Note: src_img, fwd_img & bwd_ref_img must appear consecutively in
//       kernel parameter list.
intel_sub_group_avc_ime_result_t result =
    intel_sub_group_avc_ime_evaluate_with_dual_reference(
        src_img, fwd_ref_img, bwd_ref_img, vme_sampler, payload );

// Extract IME results.
ushort sads         = intel_sub_group_avc_ime_get_inter_distortions( result );
uchar major_shape   = intel_sub_group_avc_ime_get_inter_major_shape( result );
uchar minor_shapes  = intel_sub_group_avc_ime_get_inter_minor_shapes( result );
uchar directions    = intel_sub_group_avc_ime_get_inter_directions( result );
long bi_mvs         = intel_sub_group_avc_ime_get_motion_vectors( result );
int2 bi_mvs_int     = as_int2( bi_mvs );
short2 fwd_mvs      = as_short2( bi_mvs.s0 );
short2 bwd_mvs      = as_short2( bi_mvs.s1 );
```

Example 20: IME evaluation and processing

intel_sub_group_avc_ime_result_single_reference_streamout_t
intel_sub_group_avc_ime_evaluate_with_single_reference_streamout(
image2d_t src_image,
image2d_t ref_image,
sampler_t vme_accelerator,
intel_sub_group_avc_ime_payload_t payload)

intel_sub_group_avc_ime_result_dual_reference_streamout_t
intel_sub_group_avc_ime_evaluate_with_dual_reference_streamout(
image2d_t src_image,
*image2d_t fwd*_ref_image,
*image2d_t bwd*_ref_image,
sampler_t vme_accelerator,
intel_sub_group_avc_ime_payload_t payload)

intel_sub_group_avc_ime_result_t
intel_sub_group_avc_ime_evaluate_with_single_reference_streamin(
image2d_t src_image,
image2d_t ref_image,
sampler_t vme_accelerator,
intel_sub_group_avc_ime_payload_t payload,
intel_sub_group_avc_ime_single_reference_streamin_t streamin_components)

intel_sub_group_avc_ime_result_t
intel_sub_group_avc_ime_evaluate_with_dual_reference_streamin(

```
   image2d_t src_image,
   image2d_t fwd_ref_image,
   image2d_t bwd_ref_image,
    sampler_t vme_accelerator,
   intel_sub_group_avc_ime_payload_t payload,
  intel_sub_group_avc_ime_dual_reference_streamin_t streamin_components)

intel_sub_group_avc_ime_result_single_reference_streamout_t
intel_sub_group_avc_ime_evaluate_with_single_reference_streaminout(
  image2d_t src_image,
   image2d_t ref_image,   sampler_t vme_accelerator,
  intel_sub_group_avc_ime_payload_t payload,
  intel_sub_group_avc_ime_single_reference_streamin_t streamin_components )

intel_sub_group_avc_ime_result_dual_reference_streamout_t
intel_sub_group_avc_ime_evaluate_with_dual_reference_streaminout(
  image2d_t src_image,
  image2d_t fwd_ref_image,
  image2d_t bwd_ref_image,
  sampler_t vme_accelerator,
  intel_sub_group_avc_ime_payload_t payload,
  intel_sub_group_avc_ime_dual_reference_streamin_t streamin_components )
```

Description

The VME unit internally keeps track of the best motion vectors for all shapes and sub-shapes, totaling 41 for each record of the two records (forward and backward). Once IME is finished, each record is mined for the best combination of shapes (i.e. the combination with the least distortion). The return message from the VME unit to the EU contains only the best shape combination and the remainder of the record is discarded.

For cases when the user wants to search beyond the IME search window limits (48x40 for single reference, 32x32 for dual reference) the user must call IME multiple times. Since only partial information is returned to the kernel, extracting the best shape combination across multiple calls is impossible. The best workarounds require the kernel to limit the types of shapes IME is allowed to return and then the kernel will manually merge shapes from multiple calls, which is cumbersome and suboptimal with respect to quality.

By returning more of the record to the kernel and allowing the kernel to feed in that information on subsequent calls as initialization information, the process of searching beyond IME search window size limitations is vastly improved. Now the merging of best shapes will occur inside VME and the global best shape combination is more optimized. If both records are returned in their entirety, it would be cumbersome for the implementation. A compromise yet still gain the bulk of the improvement is to stream-out only the best major shapes (9 shapes, one 16x16, two 16x8, two 8x16, and four 8x8) for both records.

IME supports searching through multiple reference regions across multiple reference images (up to 16 forward and 16 backward). This is referred to as HW assisted multi-reference searching. IME internally keeps track of the reference image by means of reference (image) identifiers. Reference identifiers are associated to pairs of forward/backward reference image parameters. Up to 16 pairs of reference pairs of reference image parameters are permitted, with the permitted values of reference identifiers ranging from 0 to 15. The reference identifiers are assigned in increasing order in which the reference image parameter pairs as declared in the OpenCL kernel parameter list as described in section [Source and Reference image parameters].

Another advantage of using the stream-out functionality is that with a single VME call, the best motion vectors and distortions can be returned for all major shapes using a single IME evaluation call which can be obtained using the result processing calls *intel_sub_group_avc_ime_get_streamout_major_shape_motion_vectors(..)* and *intel_sub_group_avc_ime_get_ streamout_major_shape_distortions(..)***.** See [4] for details for these functions. An example is shown below.

Example

```
...
// Evaluate the dual reference IME operation with streamout enable.
intel_sub_group_avc_ime_result_dual_reference_streamout_t result =
    intel_sub_group_avc_ime_evaluate_with_dual_reference_streamout(
        src_img, fwd_ref_img, bwd_ref_img, vme_sampler, payload );

// Get the best fwd/bwd 16x16 MVs. Work-item 0 has the packed MV.
uint major_fwd_mvs_16x16 =
    intel_sub_group_avc_ime_get_streamout_major_shape_motion_vectors(
        result, CLK_AVC_ME_MAJOR_16x16_INTEL, CLK_AVC_ME_MAJOR_FORWARD_INTEL );
uint major_bwd_mvs_16x16 =
    intel_sub_group_avc_ime_get_streamout_major_shape_motion_vectors(
        result, CLK_AVC_ME_MAJOR_16x16_INTEL, CLK_AVC_ME_MAJOR_BACKWARD_INTEL );

// Get the best fwd/bwd 8x8 MVs. Work-item 0 to 3 has the packed MVs in Z order.
// NOTE: Here the returned MVs are always contiguous unlike the case for
//        intel_sub_group_avc_ime_get_motion_vectors(..)   .
uint major_fwd_mvs_8x8 =
    intel_sub_group_avc_ime_get_streamout_major_shape_motion_vectors(
        result, CLK_AVC_ME_MAJOR_8x8_INTEL, CLK_AVC_ME_MAJOR_FORWARD_INTEL );
uint major_bwd_mvs_8x8 =
    intel_sub_group_avc_ime_get_streamout_major_shape_motion_vectors(
        result, CLK_AVC_ME_MAJOR_8x8_INTEL, CLK_AVC_ME_MAJOR_BACKWARD_INTEL );
```

Example 21: Using streamout to get all major shape motion vectors

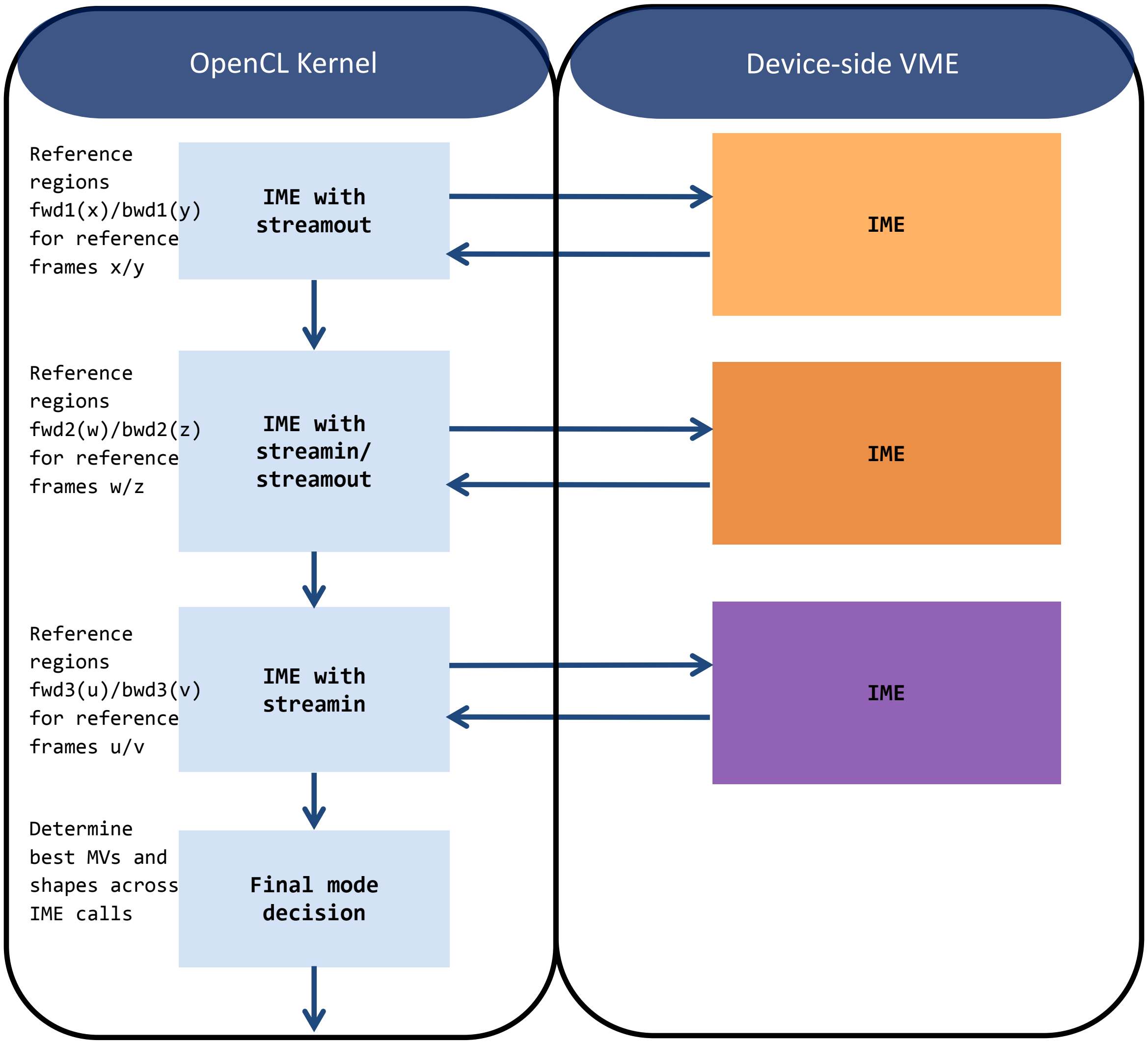

OpenCL Kernel
Device-side VME
Reference regions fwd1(x)/bwd1(y) for reference frames x/y
IME with streamout
IME
Reference regions fwd2(w)/bwd2(z) for reference frames w/z
IME with streamin/ streamout
IME
Reference regions fwd3(u)/bwd3(v) for reference frames u/v
IME with streamin
IME
Determine best MVs and shapes across IME calls
Final mode decision

Figure 21: Streamin/Streamout IME

The above figure shows an example use of the streamin/streamout feature to search multiple reference regions across multiple reference images and letting the VME unit decide the best motion vectors and their associated reference image identifiers and distortion for each pair of forward and backward reference regions.

Example

```
...
// Evaluate the dual reference IME operation with streamout enable.
intel_sub_group_avc_ime_result_dual_reference_streamout_t result =
    intel_sub_group_avc_ime_evaluate_with_dual_reference_streamout(
        src_img, fwd_ref_img, bwd_ref_img, vme_sampler, payload );

// Extract the streamout components that need to be streamin to the next IME.
intel_sub_group_avc_ime_dual_reference_streamin_t result_streamin;
result_streamin = intel_sub_group_avc_ime_get_dual_reference_streamin( result );

...
// Initialize (not shown) and configure IME to search the next reference region
payload =
  intel_sub_group_avc_ime_set_dual_reference(
    fwd_ref_offset_next_1, bwd_ref_offset_next_1,
    CLK_AVC_ME_SEARCH_WINDOW_EXHAUSTIVE_INTEL, payload
  );
...

// Evaluate the dual reference IME operation with streamin & streamout enabled
// using the results from the previous IME.
result =
    intel_sub_group_avc_ime_evaluate_with_dual_reference_streaminout(
      src_img, fwd_ref_img, bwd_ref_img, vme_sampler, payload, result_streamin );

// Extract the streamout components that need to be streamin to the next IME.
result_streamin = intel_sub_group_avc_ime_get_dual_reference_streamin( result );

...
// Initialize (not shown) and configure IME to search the next reference region
payload =
  intel_sub_group_avc_ime_set_dual_reference(
    fwd_ref_offset_next_2, bwd_ref_offset_next_2,
    CLK_AVC_ME_SEARCH_WINDOW_EXHAUSTIVE_INTEL, payload
  );
...

// Evaluate the final dual reference IME operation with streamin enable
// using the results from the previous IME.
intel_sub_group_avc_ime_result_t result_final;
result_final =
    intel_sub_group_avc_ime_evaluate_with_dual_reference_streamin(
        src_img, fwd_ref_img, bwd_ref_img, vme_sampler, payload, result_streamin );

// Get the best motion vectors across all the IME search regions.
long bi_mvs        = intel_sub_group_avc_ime_get_motion_vectors( result_final );
int2 bi_mvs_int    = as_int2( bi_mvs );
short2 fwd_mvs     = as_short2( bi_mvs.s0 );
short2 bwd_mvs     = as_short2( bi_mvs.s1 );
...
```

Example 22: Using streamin and streamout to search multiple reference regions

uint
intel_sub_group_avc_ime_get_streamout_major_shape_motion_vectors*(*
  intel_sub_group_avc_ime_result_single_reference_streamout_t result,
  uchar major_shape *)*

Description

These functions return the packed motion vectors for the input major shape from the IME single reference streamout results. The major shape parameter must be a compile-time constant. Up to 4 packed MVs are returned, one per work-item.
If the major shape is:

- 16x6, then one packed MV is returned by work-item 0
- 16x8, or 8x16, then two packed MVs than in order are returned by work-items 0 and 1
- 8x8, then four packed MVs taken in Z-order are returned by work-items 0 to 3

ushort
intel_sub_group_avc_ime_get_ streamout_major_shape_distortions*(*
  intel_sub_group_avc_ime_result_single_reference_streamout_t result,
  uchar major_shape *)*

Description

These functions return the distortions for the input major shape from the IME single reference streamout results. The major shape parameter must be a compile-time constant. Up to 4 distortions are returned, one per work-item in the same format as for motion vectors as described above.

uchar
intel_sub_group_avc_ime_get_ streamout_major_shape_reference_ids*(*
  intel_sub_group_avc_ime_result_single_reference_streamout_t result,
  uchar major_shape *)*

uchar
intel_sub_group_avc_ime_get_ streamout_major_shape_reference_ids*(*
  intel_sub_group_avc_ime_result_dual_reference_streamout_t result,
  uchar major_shape,
  uchar direction *)*

Description

These functions return the reference identifiers for the input major shape (and direction) from the IME single (dual) reference streamout results. The major shape (and direction) parameter(s) must be compile-time constant(s).

Up to 4 reference identifiers are returned, one per work-item in the same format as for motion vectors as described above.

uint
intel_sub_group_avc_ime_get_inter_reference_ids(
  intel_sub_group_avc_mce_result_t result *)*

Description

This function returns the inter MB reference identifiers in a packed integer format, with the following bits specifying the reference identifiers for the major partitions.

Bits	Penalty for Shape
3:0	Forward reference block 0
7:4	Backward reference block 0
11:8	Forward reference block 1
15:12	Backward reference block 1
19:16	Forward reference block 2
23:20	Backward reference block 2
27:24	Forward reference block 3
31:28	Backward reference block 3

Table 6: Inter macroblock block reference identifiers format

The values of each individual 4-bit reference identifier range from 0 to 15, with each value identifying the distance of ordered pair of forward/backward reference images as declared in the VME kernel parameter interface list as described in [Source and Reference image parameters]. If the dual-reference evaluation functions are not used, then the values of the backward reference identifiers are undefined.

The blocks are numbered using the Z order. For larger block sizes, the sub-block reference identifier pairs are replicated. For example, for a 16x16 block all four pairs of reference identifiers are replicated to the value of the first pair for block 0.

Note: Unless HW assisted multi-reference search was performed using the IME streamin/streamout evaluation routines, the individual 4-bit reference identifier pair values will all be the same (pointing to the same pair for forward/backward reference images).

uchar
intel_sub_group_avc_ime_get_inter_reference_interlaced_field_polarities(

```
    uint packed_reference_ids,
    uint packed_reference_parameter_field_polarities,
    intel_sub_group_avc_ime_result_t  result )
```

Description

This function returns the inter MB reference field polarities for the corresponding reference identifiers returned by intel_sub_group_avc_ime_get_inter_reference_ids(..) in a packed byte format, with the following bits specifying the reference field polarities for the major partitions.

Bit	Reference field polarity
0	Forward reference block 0
1	Forward reference block 1
2	Forward reference block 2
3	Forward reference block 3
4	Backward reference block 0
5	Backward reference block 1
6	Backward reference block 2
7	Backward reference block 3

Table 7: Packed reference block field polarities

If the dual-reference evaluation functions are not used, then the values of the backward reference field polarities are undefined. The blocks are numbered using the Z order. For larger block sizes, the sub-block reference field polarities are replicated. For example, for a 16x16 block all four pairs of reference field polarities are replicated to the value of the first pair for block 0.

The reference identifiers are provided in the format as per [Table 6: Inter macroblock block reference identifiers format]. It is typically obtained as the return value of the calling *intel_sub_group_avc_ime_get_inter_reference_ids(..)* for the corresponding IME operation result.

An extra input parameter needs to be provided to this function to specify the packed bit field of field polarities for each of the (up to 16) forward/backward interleaved pairs of reference image parameters in the same order as specified in the kernel parameter list, as used for the inter search operation. If less than 16 pairs are used then the corresponding bit field values are ignored.

Note: An important restriction is that when multiple IME operations are performed for a HW multi-assisted multi-reference search operation using the streamin/streamout capabilities, the same reference image parameter cannot be used with different polarities in the sequence of IME operations used for a same HW-assisted search operation. In other words, the field polarities for reference image parameters must be specified consistently across IME operations used in a HW assisted multi-reference search operation. In order to leverage HW assisted multi-reference search involving both fields of the same interlaced reference image object, the reference image object must be bound to two reference image kernel parameters, with one reference parameter associated with one field polarity and the other reference parameter with the other field polarity in the kernel.

REF functions

The REF functions are refinement operations performed on the results of an IME operation and computed in the CRE pipeline of the VME unit. There are two kinds of refinement operations possible – fractional motion estimation (FME) and bidirectional motion estimation (BME). Note that the REF functions do not alter the partitioning decisions made by IME, i.e. the major and minor shapes decided by IME remain unchanged. A BME REF operation may however further refine the direction decision by IME to bidirectional.

The REF evaluation can be configured with motion vector costs and shape costs as with IME evaluations. Generally the same costing mechanisms used with the associated IME will be used for REF evaluations as well.

Fractional Motion Estimation (FME)

Motion in frames often occurs in less than integer resolution. FME refines the IME full pixel search to find the best sub-pixel search result.

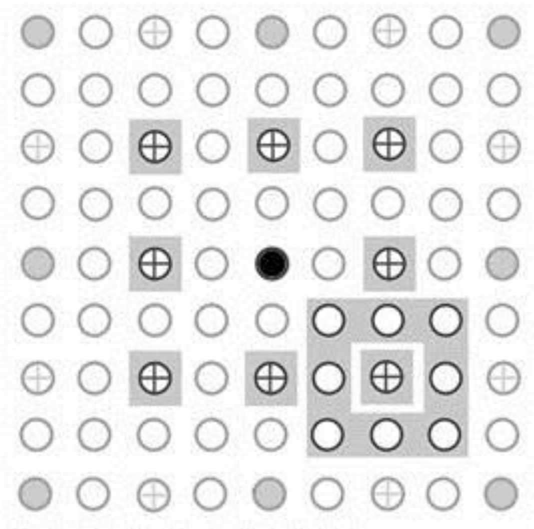


Figure 22: Sub-pixel refinement

Eight half-pixel locations are chosen around the best integer pixel location computed by IME to find the best half-pixel resolution match to the source. If enabled, 8 quarter-pixel locations around the best half pixel location are searched to find the best match to the source.

Instead of following the exact interpolation as specified by AVC, fixed 4 tap interpolation is used in the VME unit, as defined below:

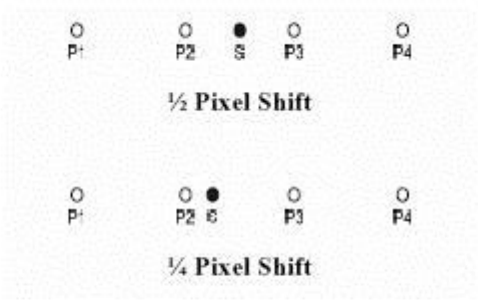


Figure 23: FME 4-tap filter

The filter computations are as shown below.

```
HALF PEL      : (-1, 5, 5, -1)/8     i.e. s = (-P1 + P2 * 5 +P3 * 5 - P4 + 4) / 8
QUARTER PEL   : (-1, 13, 5, -1)/16   i.e. c = (-P1 + P2*13 + P3 * 5 - P4 + 8)/ 16
```

The quarter-pels are actually the averages of its nearest integer and half pixel values.

Bidirectional Motion Estimation (BME)

In certain cases predicting from two reference images yields better results than from one. By encoding the images out of the (display) order, linear motion and pixel magnitude gradients can be predicted with multiple images. VME supports 5 weights of for cases where the forward and backward images are not evenly spaced from the current image. The bidirectional motion estimation process determines if the weighted average of the best matches from a forward and backward reference image yields a lesser distortion than the unidirectional distortions themselves. BME can be performed at full-pixel resolution or at sub-pixel resolution in which case an implicit FME operation is performed with the actual BME operation.

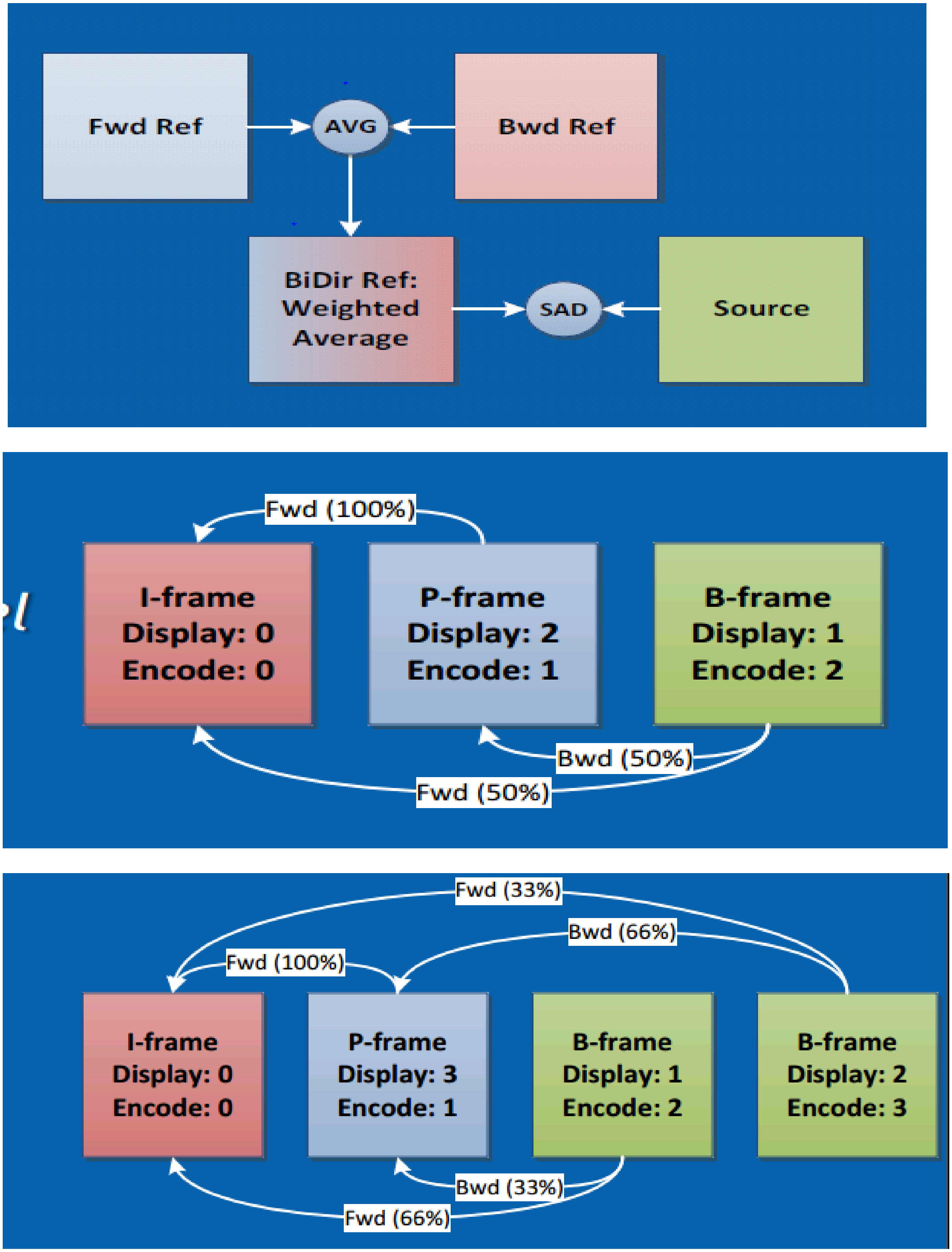
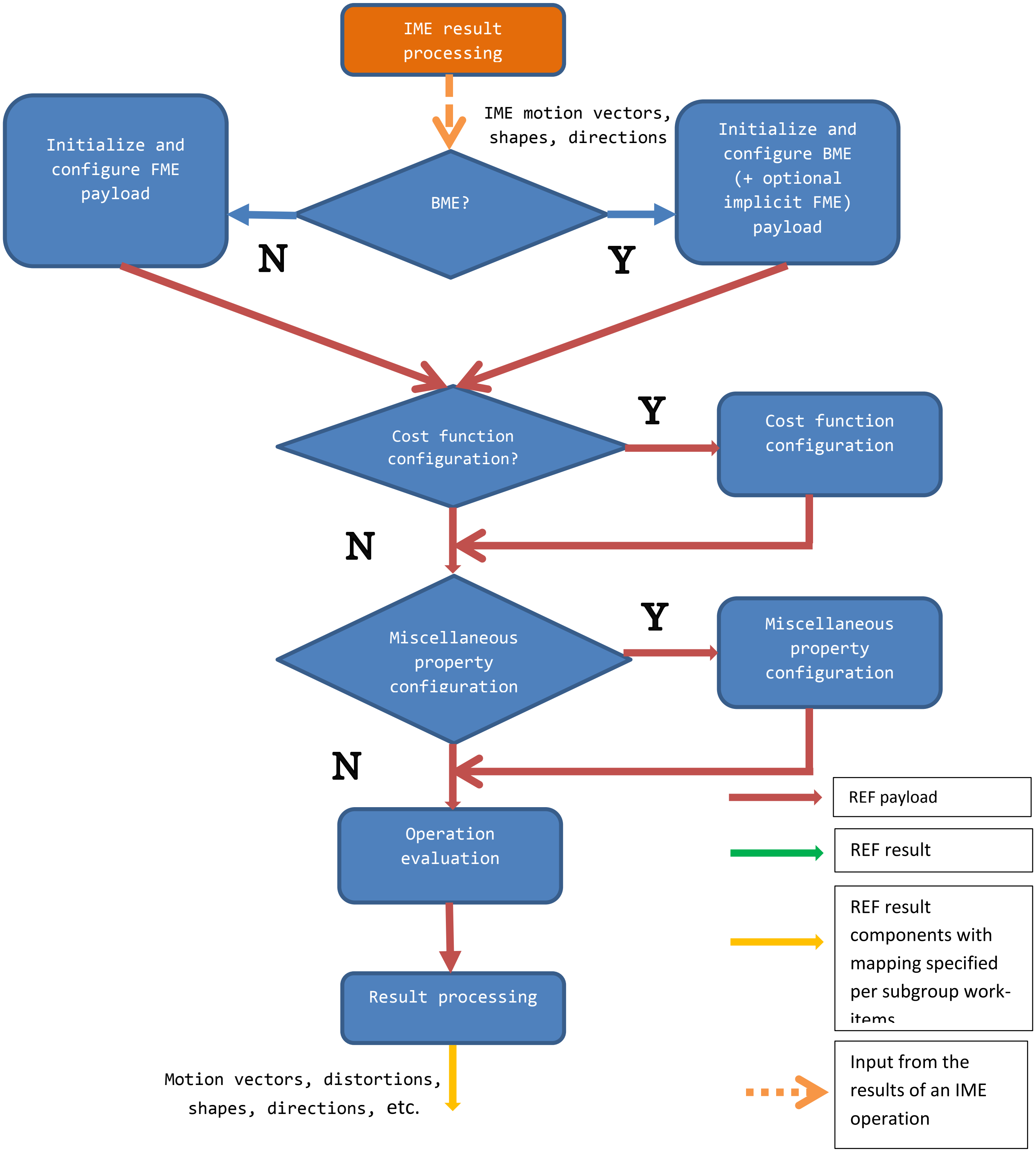

Fwd Ref
AVG
Bwd Ref
BiDir Ref: Weighted Average
SAD
Source
Fwd (100%)
I-frame Display: 0 Encode: 0
P-frame Display: 2 Encode: 1
B-frame Display: 1 Encode: 2
Bwd (50%)
Fwd (50%)
Fwd (33%)
Bwd (66%)
Fwd (100%)
I-frame Display: 0 Encode: 0
P-frame Display: 3 Encode: 1
B-frame Display: 1 Encode: 2
B-frame Display: 2 Encode: 3
Bwd (33%)
Fwd (66%)

Figure 24: Bidirectional Motion Estimation

A flowchart of the REF function phases to evaluate a REF operation is shown in Figure 25.



IME result processing
IME motion vectors, shapes, directions
BME?
Initialize and configure FME payload
N
Initialize and configure BME (+ optional implicit FME) payload
Y
Cost function configuration?
Y
Cost function configuration
N
Miscellaneous property configuration
Y
Miscellaneous property configuration
N
Operation evaluation
Result processing
Motion vectors, distortions, shapes, directions, etc.
REF payload
REF result
REF result components with mapping specified per subgroup work-items
Input from the results of an IME operation

Figure 25: REF function phase flow chart

The following section gives brief descriptions of the various function phases built-in functions and shows examples of their uses. Refer to the [4] for detailed function descriptions and their restrictions.

Initialization & Configuration phase

```
intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_fme_initialize(
   ushort2 src_coord,
   ulong motion_vectors,
   uchar major_shapes,
   uchar minor_shapes,
   uchar directions,
   uchar pixel_resolution,
   uchar sad_adjustment )

intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_bme_initialize(
   ushort2 src_coord,
   ulong motion_vectors,
   uchar major_shapes,
   uchar minor_shapes,
   uchar directions,
   uchar pixel_resolution,
   uchar bidirectional_weight,
   uchar sad_adjustment )
```

Description

These functions create and configure the payload for a FME or a BME operation.

Generally the values for the arguments for src_coord, motion_vectors, major_shapes, minor_shapes and directions can be taken directly from the results of preceding IME operation on the same macroblock. If the payload is being configured for a BME operation, then the preceding IME operation should be a dual-reference evaluation. The input arguments format match exactly with the format of the results of the corresponding IME result processing functions.

> **Note:** If the motion vector argument value is manually composed, then all sub-block MVs per its format must be replicated for each partition. For example for a 16x16 partition, all sub-block MVs must be replicated to the same MV, and for a 8x8 partition, each 8x8 must have its respective sub-block MVs replicated.

Both operations can be configured for either to use SAD or HAAR transform for the per pixel differences between the source macroblock and reference region.

The pixel resolution must be set to either half-pel or quarter-pel resolution for FME, while for BME all pixel resolutions including integer resolutions are allowed. If the BME is configured with a half-pel or quarter-pel resolution then an implicit FME operation will be performed in the CRE pipeline with the actual BME operation checking for the bidirectional mode.

The bidirectional weight parameter specifies the implicit weighting used for the forward and backward reference images. The following weighting formula is used to compute the weighted reference region used for bidirectional refined motion estimation where, alpha is one of the value of the five 5 weighting parameter enumeration values.

```
weighted_ref = ( (64 – alpha )* fwd_ref + alpha * bwd_ref  + 32 ) >> 6
```

If arbitrary weighting is required, an option is to use another OpenCL kernel to create a weighted reference images.

Example

```
    ...
    intel_sub_group_avc_ime_result_t result_ime =
      intel_sub_group_avc_ime_evaluate_with_single_reference(
          src_img, ref_img, vme_samp, payload );

    intel_sub_group_avc_ref_payload_t payload_ref =
        intel_sub_group_avc_fme_initialize(
            src_coord,
            intel_sub_group_avc_ime_get_motion_vectors( result_ime ),
            intel_sub_group_avc_ime_get_inter_major_shape( result_ime ),
            intel_sub_group_avc_ime_get_inter_minor_shapes( result_ime),
            intel_sub_group_avc_ime_get_inter_directions( result_ime ),
            CLK_AVC_ME_SUBPIXEL_MODE_QPEL_INTEL,
            CLK_AVC_ME_SAD_ADJUST_MODE_HAAR_INTEL);
```

Example 23: Basic FME initialization and configuration

```
    ...
    intel_sub_group_avc_ime_result_t result_ime =
      intel_sub_group_avc_ime_evaluate_with_dual_reference(
          src_img, ref_img, vme_samp, payload );

    // NOTE: For a BME operation, the preceding IME operation should be
    //       a dual-reference evaluation, so that bidirectional MV input
    //       are available as BME inputs.
    intel_sub_group_avc_ref_payload_t payload_bme =
        intel_sub_group_avc_bme_initialize(
            src_coord,
            intel_sub_group_avc_ime_get_motion_vectors( result_ime ),
            intel_sub_group_avc_ime_get_inter_major_shape( result_ime ),
            intel_sub_group_avc_ime_get_inter_minor_shapes( result_ime),
            intel_sub_group_avc_ime_get_inter_directions( result_ime ),
            CLK_AVC_ME_SUBPIXEL_MODE_QPEL_INTEL,
            CLK_AVC_ME_BIDIR_WEIGHT_HALF_INTEL,
            CLK_AVC_ME_SAD_ADJUST_MODE_HAAR_INTEL);
```

Example 24: Basic BME initialization and configuration

Cost configuration phase

The cost functions are a set of functions used to configure the heuristics designed in the VME unit to model the rate-distortion optimization (RDO) with the objective to minimize the bits in encoding. Ultimately the distortion is the measure used for the rate-distortion value and these cost configuration function alter the computed distortion values. The manner in which the VME unit calculates the distortion for inter estimation is shown in [Figure 15: Inter distortion calculation].

Note: Generally, if any cost configuration functions have been called for the IME operation's payload configuration, then the same cost configuration functions must also be called for configuring the corresponding REF operation's payload.

Inter cost configurations

intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_ref_set_inter_base_multi_reference_penalty(
uchar reference_base_penalty,
intel_sub_group_avc_ref_payload_t payload)

intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_ref_set_inter_direction_penalty(
uchar direction_cost,
intel_sub_group_avc_ref_payload_t payload)

```
intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_ref_set_inter_shape_penalty(
    ulong packed_shape_cost,
    intel_sub_group_avc_ref_payload_t payload )

intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_ref_set_motion_vector_cost_function(
     ulong packed_cost_center_delta,
     uint2 packed_cost_table,
     uchar cost_precision,
     intel_sub_group_avc_ref_payload_t payload )
```

Description

These functions are similar to those for IME, with the only difference being that the motion vector deltas to their cost centers have sub-pixel resolution for the motion vector configuration functions.

Miscellaneous property configuration phase

Interlaced content processing

```
intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_ref_set_source_interlaced_field_polarity(
    uchar  src_field_polarity
    intel_sub_group_avc_ref_payload_t payload )

intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_ref_set_single_reference_interlaced_field_polarity(
    uchar  ref_field_polarity,
    intel_sub_group_avc_ref_payload_t payload )

intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_ref_set_dual_reference_interlaced_image_ field_polarities(
    uchar  fwd_ref_field_polarity,
    uchar  bwd_ref_field_polarity,
    intel_sub_group_avc_ref_payload_t payload )
```

Description

These functions are similar to those for IME.

Note: If the source and reference images have been configured as interlaced for the corresponding IME operation's payload configuration, then the source and reference images must be identically configured for REF operation's payload.

Evaluation and result processing phase

intel_sub_group_avc_ref_result_t
intel_sub_group_avc_ref_evaluate_with_single_reference(
    image2d_t src_image*,*
    *image2d_t ref*_image*,*
    sampler_t vme_accelerator,
    intel_sub_group_avc_ref_payload_t payload)

intel_sub_group_avc_ref_result_t
intel_sub_group_avc_ref_evaluate_with_dual_reference(
    image2d_t src_image*,*
    *image2d_t fwd_ref*_image*,*
    *image2d_t bwd_ref*_image*,*
    sampler_t vme_accelerator,
    intel_sub_group_avc_ref_payload_t payload)

Description

These functions perform the actual REF operation in the CRE pipeline of the VME unit based on the configured payload, and where the operation latency is incurred. One version of the function is for single reference evaluations, while the other is for dual-reference evaluations. If the dual-reference version is used, the best (sub-pixel) motion vectors are returned for both the forward and backward reference images. The distortions are returned only for the best of forward, backward reference images, along with the best directions for major partitions. If the payload was configured for BME, the best direction may additionally indicate bidirectional. The result is returned in an object of an opaque type intel_sub_group_avc_ref_result_t. The result processing phase functions are required to extract the various results components.

uchar
intel_sub_group_avc_ref_get_inter_directions(
   intel_sub_group_avc_ref_result_t result *)*

Description

If the REF operation was configured for a dual-reference FME search, this function indicates whether the better match was found for the forward or backward reference image. On the other hand if it was configured for a BME search, this function additionally indicates if using bidirectional estimation produced the better match. Refer to [4] for a precise description of the interpretation of the result values.

ulong
intel_sub_group_avc_ref_get_motion_vectors(
   intel_sub_group_avc_ref_result_t result)

Description

This function is similar to the motion vector processing function for IME, with the only difference being that the motion vectors returned have sub-pixel resolution.

ushort
intel_sub_group_avc_ime_get_inter_distortions(
   intel_sub_group_avc_ime_result_t result *)*

Description

Get the inter distortions result corresponding to the MVs returned by *intel_sub_group_avc_ref_get_motion_vectors(..)*. The distortion computation is as shown in [Figure 15: Inter distortion calculation]. The inter directions result returned by *intel_sub_group_avc_ref_get_inter_directions(..)* will specify if the distortion corresponds to the forward MV, backward MV or bidirectional MVs. Up to 16 distortions are returned, one per work-item. The distortions have to be selected by their respective work-items based on the result block major and minor shapes just as for the result MVs as shown above in Figure 20.

Example

```
...
// Evaluate the REF operation with its configured payload.
// Note: src_img, fwd_img & bwd_ref_img must appear consecutively in
//       kernel parameter list.
intel_sub_group_avc_ref_result_t result =
    intel_sub_group_avc_ref_evaluate_with_dual_reference(
        src_img, fwd_ref_img, bwd_ref_img, vme_sampler, payload );

// Extract IME results.
ushort sads         = intel_sub_group_avc_ref_get_inter_distortions( result );
uchar directions    = intel_sub_group_avc_ref_get_inter_directions( result );
long bi_mvs         = intel_sub_group_avc_ime_get_motion_vectors( result );
int2 bi_mvs_int     = as_int2( bi_mvs );
short2 fwd_mvs      = as_short2( bi_mvs.s0 );
short2 bwd_mvs      = as_short2( bi_mvs.s1 );
```

Example 25: REF evaluation and result processing

intel_sub_group_avc_ref_result_t
intel_sub_group_avc_ref_evaluate_with_multi_reference(
    image2d_t src_image*,*
    uint packed_reference_ids*,*
    sampler_t vme_accelerator,
    intel_sub_group_avc_ref_payload_t payload)

intel_sub_group_avc_ref_result_t
intel_sub_group_avc_ref_evaluate_with_multi_reference(
    image2d_t src_image*,*
    uint packed_reference_ids*,*
    uchar packed_reference_field_polarities,
    sampler_t vme_accelerator,
    intel_sub_group_avc_ref_payload_t payload)

Description

These functions also perform the actual REF operation with multi-references in the CRE pipeline of the VME unit based on the configured payload, and where the operation latency is incurred. The second version of the function is to process interlaced images.

The reference identifiers (indicating unique forward/backward reference images) are specified independently for each enabled major partition. The format of the reference identifiers is described in [Table 6: Inter macroblock block reference identifiers format]. A forward[backward] reference identifier value of 'n' indicates the forward[backward] image from the 'nth' pair of forward/backward reference images, with the value of 'n' ranging from 0 to 15. If the REF operation is configured with only forward reference images then, the values of the backward reference identifiers are not used. The blocks are numbered using the Z order. For larger block sizes, the sub-block reference identifier pairs must be replicated. For example, for a 16x16 block, all four pair of reference identifiers must be replicated to the value of the first pair for block 0.

The value for the reference identifiers argument is obtained by calling intel_sub_group_avc_ime_get_inter_reference_ids(..) for the preceding IME operation's result.

The reference field polarities for forward and backward reference images are specified for each of the allowed major partitions as an integer in the format as described in [Table 7: Packed reference block field polarities].

The value for the reference identifiers polarities argument is obtained by calling intel_sub_group_avc_ime_get_inter_reference_interlaced_ field_polarities (..) for the preceding IME operation's result.

If the dual-reference evaluation functions are not used, then the values of the backward reference field polarities are not used.

The blocks are numbered using the traditional Z order. For larger block sizes, the sub-block reference field polarities are replicated. For example, for a 16x16 block all four pairs of reference field polarities are replicated to the value of the first pair for block 0.

SIC functions

The skip and intra check (SIC) functions are actually two distinct operations performed in the CRE pipeline of the VME unit – skip checks and AVC intra prediction (mode) estimation. The two operations can be performed as independent VME operations or be combined as a single VME operation by configuring the payload appropriately.

Skip Checks (SKC)

The VME unit will check the pixel distortion of a user-specified shape & motion vector combination. VME fetches the necessary pixels, performs fractional & bidirectional filtering if required, and then computes the SAD (or HAAR) between the derived reference & source, which is the result of the skip check operations. A search operation is not required as the user specifies the shape, motion vectors, and direction (forward, backward or bidirectional) as inputs, and is thus a faster operation than an IME operation.

The skip check operation can also be configured for performing an AVC 4x4 forward transform for enhancing the skip check decision. The 4x4 forward transform results are then compared one coefficient at a time against user-specified thresholds to emulate the forward quantization's zeroing effect. The count of coefficients that exceeded their threshold along with the sum of the amount exceeded is reported at an 8x8 block level, thus allowing better skip decision to be made.

The SKC evaluation can be configured with motion vector costs and shape costs as with IME or REF evaluations.

Intra Prediction Estimation (IPE)

Intra-prediction (mode) estimation is the process of determining the spatial correlation between macroblocks/blocks within the source image in order to predict the source macroblock using the edge pixels from neighboring macroblocks/blocks. The intra-prediction mode returned specify exactly which edge pixels to use and the method of prediction. IPE can be performed for either only luma, or luma and chroma pixels. If IPE is performed for chroma pixel, the source and reference images must be native NV12 images. Native NV12 images are supported through the cl_intel_planar_yuv extension.

IPE supports all Intra16x16, Intra8x8, and Intra4x4 modes (or angles): 9 modes for 4x4 and 8x8, plus 4 16x16 modes. All predictions are based on original frame pixels for quick performance, as widely adopted in HW industry. There is a known quality drop.

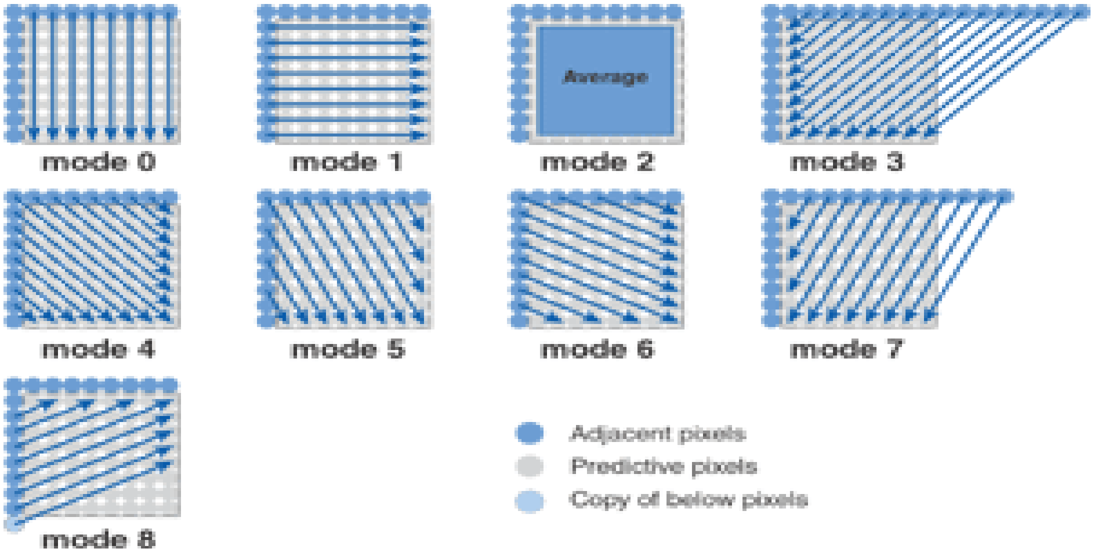


Figure 26: Intra-prediction modes for 4x4 and 8x8

Penalties can be applied to the distortion values to bias against 16x16, 8x8, or 4x4 shapes for each IPE evaluation (shape penalty). The inputs for the IPE evaluation will include the shape costs for 16x16, 8x8, or 4x4. This is similar to the shape penalty specified used for IME, FME or SICS evaluations.

A penalty can also be applied for each block\sub-block when its mode differs from the neighborhood predicted mode (mode penalty). The kernel provides the neighbor MB pixels and modes. VME does not restrict the kernel to only use original pixels. The inputs for the IPE evaluation will include the neighboring block predicted modes and the penalty term to be used the cost functions for intra prediction estimation. This is similar to the MV cost penalty used for IME, FME or SICS evaluations.

A flowchart of the SIC function phases to evaluate a SIC operation is shown in Figure 27.

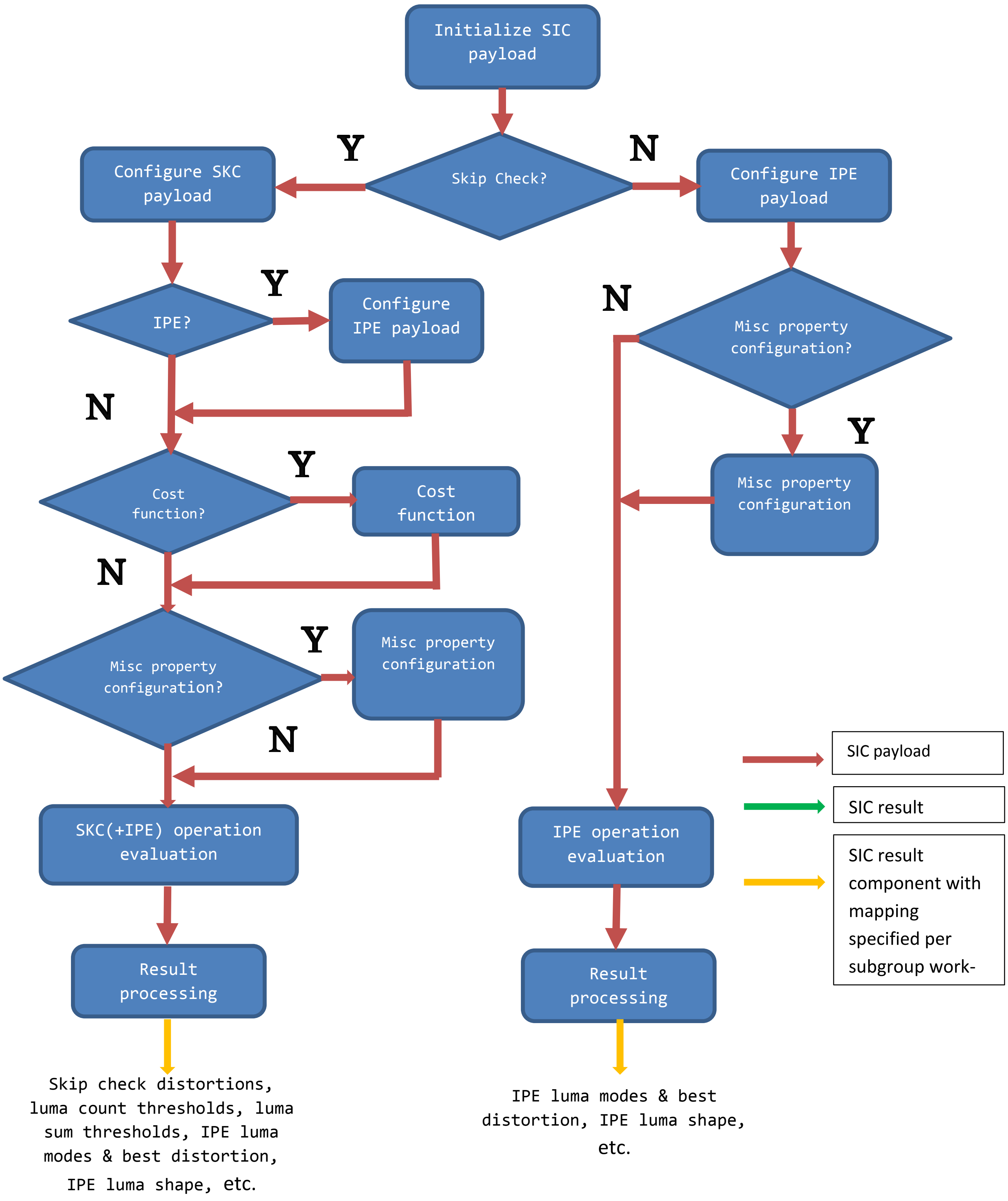

Initialize SIC payload
Skip Check?
Y
N
Configure SKC payload
Configure IPE payload
IPE?
Y
Configure IPE payload
N
Misc property configuration?
N
Y
Misc property configuration
Cost function?
Y
Cost function
N
Misc property configuration?
Y
Misc property configuration
N
SKC(+IPE) operation evaluation
IPE operation evaluation
Result processing
Result processing
Skip check distortions, luma count thresholds, luma sum thresholds, IPE luma modes & best distortion, IPE luma shape, etc.
IPE luma modes & best distortion, IPE luma shape, etc.
SIC payload
SIC result
SIC result component with mapping specified per subgroup work-

Figure 27: SIC function phases flow chart

Initialization phase

intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_initialize(
  ushort2 src_coord)

Description

Return an initialized payload for a VME SIC message in a SIMD16 kernel.

The source coordinates of the source MB in pixel units relative to the top-left corner of the source image is set in the initialized payload. If the source image is an interlaced image, then see section [Interlaced content processing] for additional information on the specification of source coordinates.

> **Note:** If the SIC operation is being configured for chroma based intra estimation in the following configuration phase, then the x and y coordinates of src_coord must be multiples of 2.

Example

```
  ushort2  src_coord;
    ...

  intel_sub_group_avc_sic_payload_t payload =
    intel_sub_group_avc_sic_initialize(
      src_coord        // source MB offset in pixel units
    );
```

Example 26: SIC payload initialization

Operation configuration phase

intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_configure_skc(
  uint skip_block_partition_type,
  uint skip_motion_vector_mask,
  ulong motion_vectors,
  uchar bidirectional_weight,
  uchar skip_sad_adjustment,
  intel_sub_group_avc_sic_payload_t payload)

Description

This function is used to configure the payload of a SKC operation. If the same payload is to be additionally configured with an IPE operation prior to evaluation, then the SKC configuration must be performed first as shown in Figure 27.

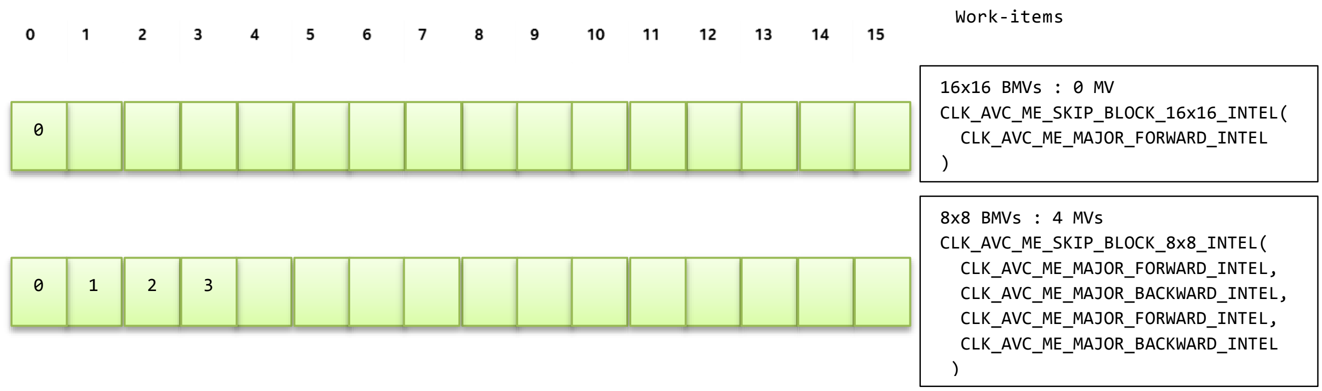
The shape (skip_block_partition_type), MV & direction combination (skip_motion_vector_mask) and MVs (motion_vectors) corresponding to the combination specified, are specified as inputs to the operation.

If the MV & direction combination specifies MVs from a forward and backward image, the bidirectional weight (bidirectional_weight) for bidirectional filtering is specified, which otherwise is ignored.

The distortion measure to be used also needs to be set as SAD or HAAR transform.

The shape can either be set to 16x16 or 8x8. No other shapes are supported for skip checks. If the MV & direction combination is a compile-time constant, then for a 16x16 shape the macro CLK_AVC_ME_SKIP_BLOCK_16x16_INTEL(DIRECTION) may be used, and for a 8x8 shape the macro CLK_AVC_ME_SKIP_BLOCK_8x8_INTEL(DIRECTION0, DIRECTION1, DIRECTION2, DIRECTION3) may be used to specify the combination for each 8x8 major partition in Z-order. Otherwise the helper configuration function intel_sub_group_avc_sic_get_motion_vector_mask(..) as described in [4] may be used.

The motion vectors are specified in a packed BMV format as described in [4]. Note that the format used here is packed format unlike the strided format used in IME and REF operations. The following figure illustrates the format.



```
16x16 BMVs : 0 MV
CLK_AVC_ME_SKIP_BLOCK_16x16_INTEL(
  CLK_AVC_ME_MAJOR_FORWARD_INTEL
)
```

```
8x8 BMVs : 4 MVs
CLK_AVC_ME_SKIP_BLOCK_8x8_INTEL(
  CLK_AVC_ME_MAJOR_FORWARD_INTEL,
  CLK_AVC_ME_MAJOR_BACKWARD_INTEL,
  CLK_AVC_ME_MAJOR_FORWARD_INTEL,
  CLK_AVC_ME_MAJOR_BACKWARD_INTEL
 )
```

Example 27: SKC motion vector input format

Example

```
ushort2  src_coord;
uchar bidirectional_weight;
uint skip_motion_vector_mask;
ulong mvs;
  ...
skip_motion_vector_mask =
    CLK_AVC_ME_SKIP_BLOCK_8x8_INTEL(
        CLK_AVC_ME_MAJOR_FORWARD_INTEL, CLK_AVC_ME_MAJOR_BACKWARD_INTEL,
        CLK_AVC_ME_MAJOR_FORWARD_INTEL, CLK_AVC_ME_MAJOR_BIDIRECTIONAL_INTEL );

intel_sub_group_avc_sic_payload_t payload =
    intel_sub_group_avc_sic_initialize( src_coord );
payload =
   intel_sub_group_avc_sic_configure_skc(
       CLK_AVC_ME_SKIP_BLOCK_PARTITION_8x8_INTEL,
       skip_motion_vector_mask,
       mvs,
       CLK_AVC_ME_BIDIR_WEIGHT_HALF_INTEL,
       CLK_AVC_ME_SAD_ADJUST_MODE_HAAR_INTEL,
       payload );
```

Example 28: SIC (SKC) payload configuration

intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_configure_ipe(
uchar luma_intra_ partition_mask,
uchar intra_neighbour_availabilty,
uchar left_edge_pixels,
uchar upper_left_corner_pixel,
uchar upper_edge_pixels,
uchar upper_right_edge_pixels,
uchar intra_sad_adjustment ,
intel_sub_group_avc_sic_payload_t payload)

Description

This function is used to configure the payload of a luma-only IPE operation. If the same payload is to be additionally configured with an SKC operation prior to evaluation, then the SKC configuration must be performed first as shown in Figure 27.

The IPE operation can be configured for 16x16, 8x8, and 4x4 block partitions. The allowed partitions mask, luma_intra_partition_mask, can be configured as described in [4. If more than one partition is selected, IPE calculates the best partition and mode combination.

The neighboring macroblock edge pixels need to be provided as inputs to the IPE configuration function. The edge pixel can be read in the kernel using the media block read operations described in [5]. Note that the same kernel image argument cannot be used across both VME and media block read/write functions. Therefore the user needs to use two separate image parameters in the OpenCL kernel, one for the VME operation and the other for the media block read/write operation. In the OCL host application the same source image object can be bound to the two separate kernel arguments.

Macroblocks along image boundaries will not have all edges available for interpolation. The available edges for the macroblocks can be set in the kernel in intra_neighbour_availabilty as described in [4].

The distortion measure to be used also needs to be set as SAD or HAAR transform.

Example

```
void sic_kernel(
    __read_only image2d_t src_vme_image,
    __read_only image2d_t ref_image,
    __read_only image2d_t src_read_image,
    ushort2               src_coord
)
{
    ...

    // Initialize the MB neighborhood mask, intraEdges.
    uint intraEdges,_leftEdge, leftUpperPixel, upperEdge;
    intraEdges =
        CLK_AVC_ME_INTRA_NEIGHBOR_LEFT_MASK_ENABLE_INTEL       |
        CLK_AVC_ME_INTRA_NEIGHBOR_UPPER_MASK_ENABLE_INTEL      |
        CLK_AVC_ME_INTRA_NEIGHBOR_UPPER_LEFT_MASK_ENABLE_INTEL |
        CLK_AVC_ME_INTRA_NEIGHBOR_UPPER_RIGHT_MASK_ENABLE_INTEL;

      // If this is a left-edge MB, then disable left edges.
      if( ... ) {
        intraEdges &= ~CLK_AVC_ME_INTRA_NEIGHBOR_LEFT_MASK_ENABLE_INTEL;
        intraEdges &= ~CLK_AVC_ME_INTRA_NEIGHBOR_UPPER_LEFT_MASK_ENABLE_INTEL;
      }
      // If this is a right edge MB then disable right edges.
      if( ... ) {
        intraEdges &= ~CLK_AVC_ME_INTRA_NEIGHBOR_UPPER_RIGHT_MASK_ENABLE_INTEL;
      }
      // If this is a top-edge MB, then disable top edges.
      if( ... ) {
        intraEdges &= ~CLK_AVC_ME_INTRA_NEIGHBOR_UPPER_LEFT_MASK_ENABLE_INTEL;
        intraEdges &= ~CLK_AVC_ME_INTRA_NEIGHBOR_UPPER_RIGHT_MASK_ENABLE_INTEL;
        intraEdges &= ~CLK_AVC_ME_INTRA_NEIGHBOR_UPPER_MASK_ENABLE_INTEL;
      }
```

```
        // Read left edge.
        int2 edgeCoord;
        edgeCoord.x = srcCoord.x - 4; edgeCoord.y = srcCoord.y;
        uint leftEdgeDW =
            intel_sub_group_media_block_read_ui( edgeCoord, 1, 16, src_read_image );
        leftEdge = as_uchar4( leftEdgeDW ).s3;


        // Read upper left corner.
        edgeCoord.x = srcCoord.x - 4; edgeCoord.y = srcCoord.y - 1;
        uint leftUpperPixelDW =
            intel_sub_group_media_block_read_ui( edgeCoord, 1, 16, src_read_image );
        leftUpperPixel = as_uchar4( leftUpperPixelDW ).s3;
        leftUpperPixel = intel_sub_group_shuffle( leftUpperPixel, 0 );


        // Read upper edge.
        edgeCoord.x = srcCoord.x; edgeCoord.y = srcCoord.y - 1;
        upperEdge =
            intel_sub_group_media_block_read_uc( edgeCoord, 16, 1, src_read_image );

        // Read upper right edge.
        edgeCoord.x = srcCoord.x + 16; edgeCoord.y = srcCoord.y - 1;
        upperRightEdge =
            intel_sub_group_media_block_read_uc( edgeCoord, 16, 1, src_read_image );

        // Initialize a SIC operation.
        intel_sub_group_avc_sic_payload_t payload =
            intel_sub_group_avc_sic_initialize( src_coord );

        // Configure an optional SKC operation.
        payload =
           intel_sub_group_avc_sic_configure_skc( ... );

        // Configure an IPE operation with the neighboring edges.
        payload =
            intel_sub_group_avc_sic_configure_ipe(
                intraPartMask, intraEdges,
                leftEdge, leftUpperPixel, upperEdge, upperRightEdge,
                CLK_AVC_ME_SAD_ADJUST_MODE_HAAR_INTEL, payload );
        ...
```

Example 29: SIC (IPE) payload configuration

intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_configure_ipe(
  uchar luma_intra_ partition_mask,
  uchar intra_neighbour_availabilty,
  uchar left_edge_pixels,
  uchar upper_left_corner_pixel,
  uchar upper_edge_pixels,
  uchar upper_right_edge_pixels,

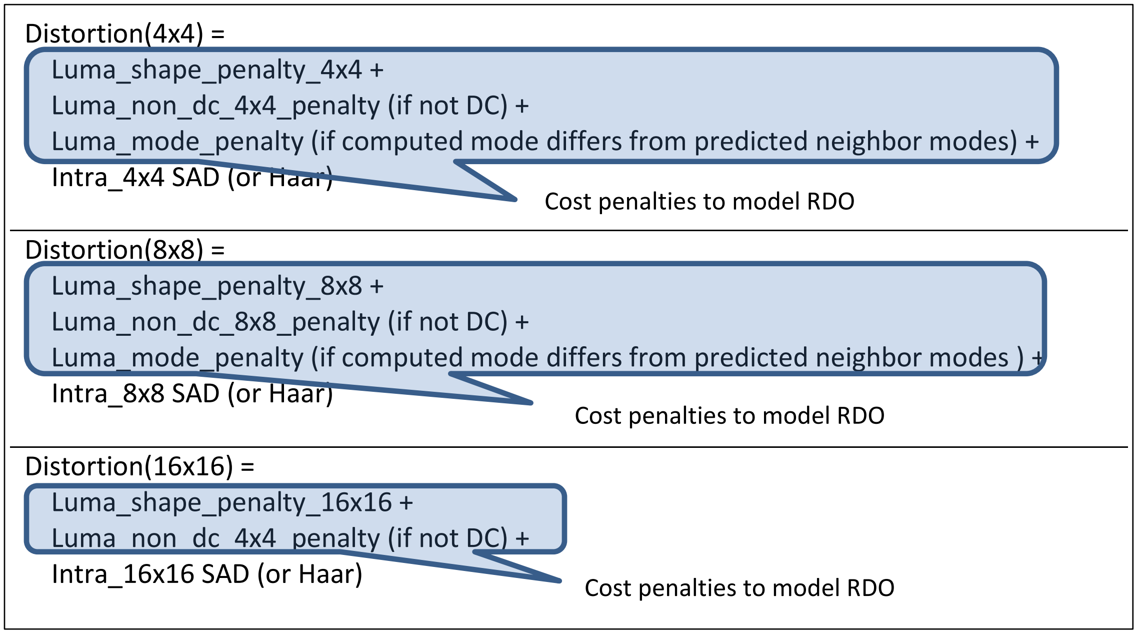
```
    ushort left_edge_chroma_pixels,
    ushort upper_left_corner_chroma_pixel,
    ushort upper_edge_chroma_pixels,
    uchar intra_sad_adjustment ,
    intel_sub_group_avc_sic_payload_t payload )
```

Description

This function is used to configure the payload of a luma and chroma IPE operation. If the same payload is to be additionally configured with an SKC operation prior to evaluation, then the SKC configuration must be performed first as shown in Figure 27.

Cost configuration phase

The cost functions are a set of functions used to configure the heuristics designed in the VME unit to model the rate-distortion optimization (RDO) with the objective to minimize the bits in encoding. Ultimately the distortion is the measure used for the rate-distortion value and these cost configuration function alter the computed distortion values. The manner in which the VME unit calculates the distortion for intra estimation is shown in [Figure 28: Intra luma distortion calculation] and [Figure 29: Intra chroma distortion calculation].


Distortion(4x4) =
Luma_shape_penalty_4x4 +
Luma_non_dc_4x4_penalty (if not DC) +
Luma_mode_penalty (if computed mode differs from predicted neighbor modes) +
Intra_4x4 SAD (or Haar)

Cost penalties to model RDO

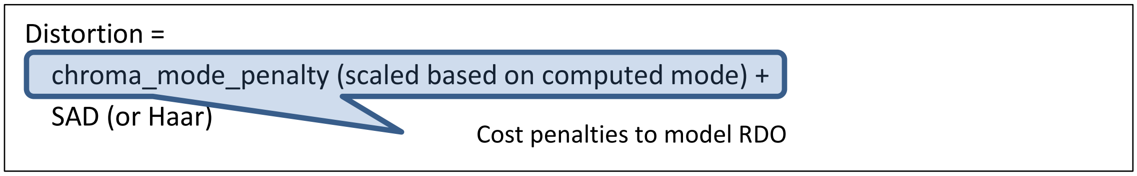
Distortion(8x8) =
Luma_shape_penalty_8x8 +
Luma_non_dc_8x8_penalty (if not DC) +
Luma_mode_penalty (if computed mode differs from predicted neighbor modes) +
Intra_8x8 SAD (or Haar)

Cost penalties to model RDO

Distortion(16x16) =
Luma_shape_penalty_16x16 +
Luma_non_dc_4x4_penalty (if not DC) +
Intra_16x16 SAD (or Haar)

Cost penalties to model RDO

Figure 28: Intra luma distortion calculation


Distortion =
chroma_mode_penalty (scaled based on computed mode) +
SAD (or Haar)

Cost penalties to model RDO

Figure 29: Intra chroma distortion calculation

Good default values to use are provided by the API, by means of built-in functions, to configure the various cost penalties. These default values are derived based on analysis done across a wide range of video content, and can be obtained per slice type and the quantization parameter (Qp) value. The API, however, does allow for the provision to override the defaults values.

Inter cost configurations

```
intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_set_inter_base_multi_reference_penalty(
    uchar reference_base_penalty,
    intel_sub_group_avc_sic_payload_t payload )

intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_ref_set_inter_direction_penalty(
     uchar direction_cost,
     intel_sub_group_avc_ref_payload_t payload )

intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_ref_set_inter_shape_penalty(
     ulong packed_shape_cost,
     intel_sub_group_avc_ref_payload_t payload )

intel_sub_group_avc_ref_payload_t
intel_sub_group_avc_ref_set_motion_vector_cost_function(
      ulong packed_cost_center_delta,
      uint2 packed_cost_table,
      uchar cost_precision,
      intel_sub_group_avc_ref_payload_t payload )
```

Description

These functions are identical to the corresponding configuration functions for REF. However it is valid to call this only if the payload had been previously configured for an SKC operation.

Intra cost configurations

Intra (luma) shape cost configuration

```
uint
intel_sub_group_avc_mce_get_default_intra_luma_shape_penalty(
    uchar slice_type,
```

uchar qp)

intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_set_intra_luma_shape_penalty(
uint packed_shape_cost,
intel_sub_group_avc_sic_payload_t payload)

Description

This function updates the input payload to specify the shape penalty for intra motion estimation. The shape penalty is specified as an unsigned integer value with the bits specifying the shape penalty in U4U4 format as shown in the table below.

Bits	Penalty for Shape
7:0	Must be zero
15:8	16x16
23:16	8x8
31:24	4x4

Table 8: Intra packed shape penalty format

The U4U4 decoded integer values must bit fit within 12 bits.

Example

```
...
// Get a good default shape penalties to use.
uint shape_penalties =
  intel_sub_group_avc_mce_get_default_intra_luma_shape_penalty(
      slice_type, qp );
// Update the payload with the shape cost penalties.
payload =
  intel_sub_group_avc_sic_set_intra_luma_shape_penalty(
      shape_penalties, payload );
```

Figure 30: Intra luma shape penalty configuration

The API provides good default values to use for shape costs, but provisions exist for providing custom values. A rough model for specifying custom values is provided below.

```
   lambda = pow(2, (qp-12)/6);
   bias = 2.0;
   union {
      uint     raw;
      struct {
         uchar s0;          // Must be zero
         uchar s1;          // 16x16
         uchar s2;          // 8x8
         uchar s3;          // 4x4
      } bits;
   } packedShapeCost;

   packedShapeCost.raw = 0;

   float IntraShapeCost[3] =
   {
                    // PSLICE
      10.0,         // 16x16
      14.0,         // 8x8
      35.0,         // 4x4
   };

   packedShapeCost.bits.s1 = min( ConvertToU4U4((U16)( lambda *
      IntraShapeCost[ INTRA_16x16 ]  * bias )), 0x8f );
   packedShapeCost.bits.s2 = min( ConvertToU4U4((U16)( lambda *
      IntraShapeCost[ INTRA_8x8 ]  * bias )), 0x8f );
   packedShapeCost.bits.s3 = min( ConvertToU4U4((U16)( lambda *
      IntraShapeCost[ INTRA_4x4 ]  * bias )), 0x8f );
   payload_t =  intel_sub_group_avc_mce_set_intra_shape_penalty(
      packedShapeCost.raw, payload )
```

Figure 31: Sample model for intra shape penalty

Intra luma mode cost configuration

uchar

intel_sub_group_avc_mce_get_default_intra_luma_mode_penalty(

uchar slice_type,

uchar qp)

uint

intel_sub_group_avc_mce_get_default_non_dc_luma_intra_penalty(*void*)

intel_sub_group_avc_sic_payload_t

intel_sub_group_avc_sic_set_intra_luma_mode_cost_function(

uchar luma_mode_penalty,

uint luma_packed_neighbor_modes,

uint luma_packed_non_dc_penalty,

intel_sub_group_avc_sic_payload_t payload)

Description

This function updates the input payload to specify the mode penalty, neighbor intra modes and non-dc mode penalty for intra luma mode estimation. The mode penalty will be applied to the estimated luma mode if it differs from its predicted luma mode (based on its neighbor intra modes). It is specified in U4U4 format and must bit in 10 bits.

The neighbor modes specify the values of the already computed top and left neighbor modes for the bordering 4x4 blocks, with the 4x4 block numbered in Z-order (see below).

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Figure 32: Z-order layout

The neighbor intra modes are specified using an unsigned integer with the bits as described in the table below specifying the respective neighbor modes.

Bits	Neighbor mode
3:0	Left neighbor block 5
7:4	Left neighbor block 7
11:8	Left neighbor block D
15:12	Left neighbor block F
19:16	Top neighbor block A
23:20	Top neighbor block B
27:24	Top neighbor block E
31:28	Top neighbor block F

Table 9: Intra packed luma neighbor mode format

The flat penalty for non-dc modes will be applied for any computed non-DC luma mode for each of the 16x16, 8x8, and 4x4 shapes, with the bits as described in the table below specifying the respective penalties.

Bits	Non-DC Penalty
7:0	Intra16x16 non-dc penalty
15:8	Intra8x8 non-dc penalty
23:16	Intra4x4 non-dc penalty
31:24	Must be zero

Table 10: Intra packed non-dc penalty format

Example

```
bool do_mode_costing = false;
bool do_left_mode_costing = false; bool do_top_mode_costing = false;

// Gather the neighbor modes…
ulong left_modes = 0, top_modes = 0; uchar left_shape = 0, top_shape = 0;
if( intra_edges & CLK_AVC_ME_INTRA_NEIGHBOR_LEFT_MASK_ENABLE_INTEL ) {
    left_modes = ...;  left_shape = ...; do_left_mode_costing = true;
}

// Read top neighbor modes and shape an determine if we can do mode costing.
if( intra_edges & CLK_AVC_ME_INTRA_NEIGHBOR_UPPER_MASK_ENABLE_INTEL ) {
   top_modes = ...; top_shape = ...; do_top_mode_costing = true;
}
do_mode_costing = do_left_mode_costing & do_top_mode_costing;

uint neighbor_luma_modes = 0;

if( do_mode_costing ) {

   // Compose neighbor modes from left neighbor.
   if( left_shape == CLK_AVC_ME_INTRA_16x16_INTEL ) {
      neighbor_luma_modes |= (left_modes & 0xF) << 0;
      neighbor_luma_modes |= (left_modes & 0xF) << 4;
      neighbor_luma_modes |= (left_modes & 0xF) << 8;
      neighbor_luma_modes |= (left_modes & 0xF) << 12;
   } else if( left_shape == CLK_AVC_ME_INTRA_8x8_INTEL ) {
      neighbor_luma_modes |= ((left_modes >> 16) & 0xF) << 0;
      neighbor_luma_modes |= ((left_modes >> 16) & 0xF) << 4;
      neighbor_luma_modes |= ((left_modes >> 48) & 0xF) << 8;
      neighbor_luma_modes |= ((left_modes >> 48) & 0xF) << 12;
   } else {
      neighbor_luma_modes |= ((left_modes >> 20) & 0xF) << 0;
      neighbor_luma_modes |= ((left_modes >> 28) & 0xF) << 4;
      neighbor_luma_modes |= ((left_modes >> 52) & 0xF) << 8;
      neighbor_luma_modes |= ((left_modes >> 60) & 0xF) << 12;
   }

   // Compose neighbor modes from top neighbor.
   if( top_shape == CLK_AVC_ME_INTRA_16x16_INTEL ) {
      neighbor_luma_modes |= (top_modes & 0xF) << 0;
      neighbor_luma_modes |= (top_modes & 0xF) << 4;
      neighbor_luma_modes |= (top_modes & 0xF) << 8;
      neighbor_luma_modes |= (top_modes & 0xF) << 12;
   } else if( top_shape == CLK_AVC_ME_INTRA_8x8_INTEL ) {
      neighbor_luma_modes |= ((top_modes >> 32) & 0xF) << 16;
      neighbor_luma_modes |= ((top_modes >> 32) & 0xF) << 20;
      neighbor_luma_modes |= ((top_modes >> 48) & 0xF) << 24;
      neighbor_luma_modes |= ((top_modes >> 48) & 0xF) << 28;
   } else {
      neighbor_luma_modes |= ((top_modes >> 40) & 0xFF) << 16;
      neighbor_luma_modes |= ((top_modes >> 56) & 0xFF) << 24;
   }
```

```
// Get good default mode penalties to use…
uchar mode_penalty =
   intel_sub_group_avc_mce_get_default_intra_luma_mode_penalty(slice_type, qp );
uint nondc_penalty =
   intel_sub_group_avc_mce_get_default_non_dc_luma_intra_penalty();

// Update the payload with the mode penalties.
payload =
  intel_sub_group_avc_sic_set_intra_luma_mode_cost_function(
      mode_penalty, neighbor_luma_modes, nondc_penalty, payload );
```

Example 30: Luma mode cost configuration

The API provides good default values to use for the mode penalty, but provisions exist for providing custom values. A rough model for specifying custom values is provided below.

```
lambda = pow(2, (qp-12)/6);
bias = 2.0;

uchar mode_penalty = 0;

double IntraModeCostISlice[52] =
{
  8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, //QP=[0 ~12]
  8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 7.0, 7.0, 7.0, //QP=[13~25]
  7.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 4.0, 4.0, 4.0, 4.0, //QP=[26~38]
  4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 3.0, 3.0, 3.0, 3.0, 3.0, //QP=[39~51]
};

float value = ( SliceType == INTRASLICE )? IntraModeCost[ qp ] : 4.0;
mode_penalty = min( ConvertToU4U4((U16)( lambda * value * bias )), 0x6f );
...
payload =
  intel_sub_group_avc_sic_set_intra_luma_mode_cost_function(
      mode_penalty, neighbor_luma_modes, nondc_penalty, payload );
```

Figure 33: Sample model luma mode penalty

Intra chroma mode cost configuration

uchar
intel_sub_group_avc_mce_get_default_intra_chroma_mode_base_penalty(*void*)

intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_set_intra_chroma_mode_cost_function(
    uchar chroma_mode_penalty,
    intel_sub_group_avc_sic_payload_t payload)

Description

This function updates the input payload to specify the base penalty to be applied to the computed chroma modes. This penalty is in U4U4 format and its decoded integer value must fit in 12 bits.

The base penalty is scaled based on the computed mode as described in the table below.

Chroma Mode	Scaled Penalty
DC	0x
HORZ	1x
VERT	1x
PLANE	2x

Table 11: Chroma mode base penalty scale factors

```
   // Get good default base penalties to use…
   uchar chroma_mode_penalty =
      intel_sub_group_avc_mce_get_default_intra_chroma_mode_base_penalty();

   // Update the payload with the mode penalties.
   payload =
     intel_sub_group_avc_sic_set_intra_chroma_mode_cost_function(
         chroma_mode_penalty, payload );
```

Figure 34: Intra chroma mode cost configuration

Miscellaneous property configuration phase

Interlaced content processing

intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_set_source_interlaced_field_polarity(
    uchar *src*_field_polarity
    intel_sub_group_avc_sic_payload_t payload)

intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_set_single_reference_interlaced_field_polarity(
    uchar ref_field_polarity,
    intel_sub_group_avc_sic_payload_t payload)

intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_set_dual_reference_ interlaced_field_polarities(
    uchar *fwd*_ref_field_polarity,
    uchar *bwd*_ref_field_polarity,

```
    intel_sub_group_avc_sic_payload_t payload )
```

Description

These functions are similar to those for IME.

Skip check forward transform

```
intel_sub_group_avc_sic_payload_t
intel_sub_group_avc_sic_set_skc_forward_transform_enable(
    ulong  packed_sad_coefficients,
    intel_sub_group_avc_sic_payload_t payload )
```

Description

If the SIC payload was configured for a SKC operation, this function enables extra computations in the CRE pipeline to enhance the skip decision to include an accurate AVC 4x4 forward transform. This feature is in addition to the previous SAD or HAAR skip estimation. The results of the forward transform are compared one coefficient at a time against a user-specified threshold, in the input argument packed_sad_coefficients, to emulate the forward quantization's zeroing effect. The user is returned the count of coefficients that exceeded their threshold along with a sum of the amount exceeded, both grouped at the 8x8 block level.

The SAD coefficient threshold matrix for a 4x4 transform as shown in the table below is packed into a 64-bit integer. The values programmed are dependent upon the quantization parameter used for the macroblock. The low 16 bits contains the larger DC threshold. The coefficient thresholds for the remaining 6 AC thresholds in the order of increasing frequency are provided by the successive 8-bit bit ranges.

0 (DC)	1 (AC)	2 (AC)	3 (AC)
1 (AC)	2 (AC)	3 (AC)	4 (AC)
2 (AC)	3 (AC)	4 (AC)	5 (AC)
3 (AC)	4 (AC)	5 (AC)	6 (AC)

Table 12: Packed SAD coefficient thresholds format

Example

```
  intel_sub_group_avc_sic_payload_t payload;
  ...
  union {
     struct {
       ushort dc;
       uchar  ac1;
       uchar  ac2;
       uchar  ac3;
       uchar  ac4;
       uchar  ac5;
       uchar  ac6;
     } x;
     ulong y;
   } packed_sad_coefficients;

   packed_sad_coefficients.x.dc  = dc_threshold;
   packed_sad_coefficients.x.ac1 = ac1_threshold;
   packed_sad_coefficients.x.ac2 = ac2_threshold;
   packed_sad_coefficients.x.ac3 = ac3_threshold;
   packed_sad_coefficients.x.ac4 = ac4_threshold;
   packed_sad_coefficients.x.ac5 = ac5_threshold;
   packed_sad_coefficients.x.ac6 = ac6_threshold;

   payload =
     intel_sub_group_avc_sic_set_skc_forward_transform_enable(
         packed_sad_coefficients.y,
         payload );
   ...
```

Example 31: Forward transform threshold enabling

Evaluation and result processing phase

intel_sub_group_avc_sic_result_t
intel_sub_group_avc_sic_evaluate_ipe(
   image2d_t src_image*,*
   sampler_t vme_accelerator,
   intel_sub_group_avc_sic_payload_t payload)

Note: If the SIC operation is being configured for chroma based intra estimation, then the argument for src_image must be a native NV12 image.

Description

This function performs the actual IPE operation in the CRE pipeline of the VME unit based on the configured payload and where the operation latency is incurred. This evaluation function should be called when the payload is configured to perform only an IPE operation as shown in [Figure 27].

uchar
intel_sub_group_avc_sic_get_ipe_luma_shape(
    intel_sub_group_avc_sic_result_t result)

Description

This function returns the block partitioning recommendation from the VME unit. The macroblock is evenly split into 4x4, 8x8 blocks or retained as a 16x16 block. Refer to [4] for a precise description of the interpretation of the result values.

ushort
intel_sub_group_avc_sic_get_best_ipe_luma_distortion(
    intel_sub_group_avc_sic_result_t result)

Description

This function returns the best intra distortion from the VME unit for the shape recommendation returned by *intel_sub_group_avc_sic_get_ipe_luma_shape*(…). Note that a single scalar value is returned by this function which represents the distortion value for all sub-blocks.

ulong
intel_sub_group_avc_sic_get_packed_ipe_luma_modes(
    intel_sub_group_avc_sic_result_t result)

Description

This function returns the luma modes for the all blocks in the macroblock in bit-fields within a packed 64-bit integer. The blocks in the macroblocks correspond to the shape decision returned by *intel_sub_group_avc_sic_get_ipe_luma_shape*(…).Refer to [4] for a precise description of the bit-fields.

Example

```
...
// Configure an IPE operation with the neighboring edges.
payload =
    intel_sub_group_avc_sic_configure_ipe(
        intraPartMask, intraEdges,
        leftEdge, leftUpperPixel, upperEdge, upperRightEdge,
        CLK_AVC_ME_SAD_ADJUST_MODE_HAAR_INTEL, payload );

// Evaluate the SIC IPE operation with its configured payload.
intel_sub_group_sic_ref_result_t result =
    intel_sub_group_avc_sic_evaluate_ipe(
        srcImg, vme_sampler, payload );

// Extract SIC IPE results.
uchar shape = intel_sub_group_avc_sic_get_ipe_luma_shape( result );
ushort dist = intel_sub_group_avc_sic_get_best_ipe_luma_distortion( result );
ulong modes = intel_sub_group_avc_sic_get_packed_ipe_luma_modes( result );
```

Example 32: SIC IPE evaluation and result processing

intel_sub_group_avc_sic_result_t
intel_sub_group_avc_sic_evaluate_with_single_reference(
    image2d_t src_image*,*
    *image2d_t ref*_image*,*
    sampler_t vme_accelerator,
    intel_sub_group_avc_sic_payload_t payload)

intel_sub_group_avc_sic_result_t
intel_sub_group_avc_sic_evaluate_with_dual_reference(
    image2d_t src_image*,*
    *image2d_t fwd_ref*_image*,*
    *image2d_t bwd_ref*_image*,*
    sampler_t vme_accelerator,
    intel_sub_group_avc_sic_payload_t payload)

Note: If the SIC operation is being configured for chroma based intra estimation, then the argument for src_image must be a native NV12 image.

Description

These functions perform the actual SIC (SKC or SIC+IPE) operation in the CRE pipeline of the VME unit based on the configured payload and where the operation latency is incurred. One version of the function is for single reference SKC evaluations, while the other is for dual-reference SKC evaluations.

This evaluation function should be called when the payload is configured to perform an SKC operation (an optionally along with a combined IPE operation) as shown in [Figure 27]. The result processing phase functions are required to extract the various results components.

ushort
intel_sub_group_avc_sic_get_inter_distortions(
intel_sub_group_avc_sic_result_t result *)*

Description

Get the inter distortions result corresponding to the shape & motion vector combination specified in the input payload. The distortions returned are after fractional and bidirectional filtering is performed as necessary. The distortion computation is as shown in **Error! Reference source not found.**. Up to 16 istortions are returned, one per work-item. The distortions have to be selected by their respective work-items based on the result block major and minor shapes just as for the result MVs as shown above in Figure 20.

uint
intel_sub_group_avc_sic_get _packed_skc_luma_count_threshold(
intel_sub_group_avc_sic_result_t result)

ulong
intel_sub_group_avc_sic_get _packed_skc_luma_sum_threshold(
intel_sub_group_avc_sic_result_t result)

Description

These functions return the enhanced skip check results for SKC operations with their payloads configured with *intel_sub_group_avc_sic_set_ skc_forward_transform_enable(..)*. It returns the count and sum of luma coefficient components that exceeded their transform thresholds from the SIC result for each 8x8 partition in Z-order in bit-fields within a packed 64-bit integer. Refer to [4] for a precise description of the bit-fields.

Example

```
...
// Configure a SKC operation.
payload =
   intel_sub_group_avc_sic_configure_skc(
       CLK_AVC_ME_SKIP_BLOCK_PARTITION_8x8_INTEL,
       skip_motion_vector_mask, mvs,
       CLK_AVC_ME_BIDIR_WEIGHT_HALF_INTEL,
       CLK_AVC_ME_SAD_ADJUST_MODE_HAAR_INTEL,
       payload );

// Configure an IPE operation with the neighboring edges.
payload =
    intel_sub_group_avc_sic_configure_ipe(
        intraPartMask, intraEdges,
        leftEdge, leftUpperPixel, upperEdge, upperRightEdge,
        CLK_AVC_ME_SAD_ADJUST_MODE_HAAR_INTEL, payload );

// Configure the SKC operation to perform an enhanced skip check with
// an accurate AVC 4x4 forward transform and compare its results with
// the input SAD coefficients to emulate the quantizer’s zeroing effect.
payload =
    intel_sub_group_avc_sic_set_skc_forward_transform_enable(
        packed_sad_coefficients, payload );

// Evaluate the SIC IPE operation with its configured payload.
intel_sub_group_sic_ref_result_t result =
    intel_sub_group_avc_sic_evaluate_with_single_reference(
        src_img, ref_img, vme_sampler, payload );

// Extract SIC enchanced SKC results.
ushort dist = intel_sub_group_avc_sic_get_inter_distortions( result );
uint count = intel_sub_group_avc_sic_get _packed_skc_luma_count_threshold( result );
ulong sum = intel_sub_group_avc_sic_get _packed_skc_luma_sum_threshold( result );

// Extract SIC IPE results.
uchar shape = intel_sub_group_avc_sic_get_ipe_luma_shape( result );
ushort dist = intel_sub_group_avc_sic_get_best_ipe_luma_distortion( result );
ulong modes = intel_sub_group_avc_sic_get_packed_ipe_luma_modes( result );
```

Example 33: SIC SKC+IPE evaluation and result processing

intel_sub_group_avc_sic_result_t
intel_sub_group_avc_sic_evaluate_with_multi_reference(
    image2d_t src_image,
    uint packed_reference_ids,
    sampler_t vme_accelerator,
    intel_sub_group_avc_sic_payload_t payload)

intel_sub_group_avc_sic_result_t
intel_sub_group_avc_sic_evaluate_with_multi_reference(

```
    image2d_t src_image,
    uint packed_reference_ids,
    uchar packed_reference_field_polarities,
    sampler_t vme_accelerator,
    intel_sub_group_avc_sic_payload_t payload )
```

Description

These functions also perform the actual SIC (SKC or SIC+IPE)   operation with multi-references in the CRE pipeline of the VME unit based on the configured payload, and where the operation latency is incurred. The second version of the function is to process interlaced images.

The description of the function arguments is similar to their corresponding REF equivalents as described in [Evaluation and result processing phase].

These evaluation functions should be called when the payload is configured to perform a multi-reference SKC operation (an optionally along with a combined IPE operation) as shown in [Figure 27]. The result processing phase functions are required to extract the various results components.

VME with no macroblock dependencies (fully parallel VME)

VME operations may be performed on macroblocks at a frame-level in a fully parallel manner with no macroblock dependencies. This are often used as pre-processing steps in the encode pipeline. An example of such a pre-processing step is getting good initial reference search window offsets (a.k.a. predictors) for the later VME stages, that perform VME operations considering the neighboring macroblock motion estimation results in the actual encode loop. Another application is for hybrid GPU accelerated CPU encoders which use VME to gather the motion vectors and distortion data for all partition sizes at a frame-level in order for a later CPU stage to make the actual VME decisions in the encode loop.

The following figure show an application of frame-level VME (in red) in a general encode pipeline.

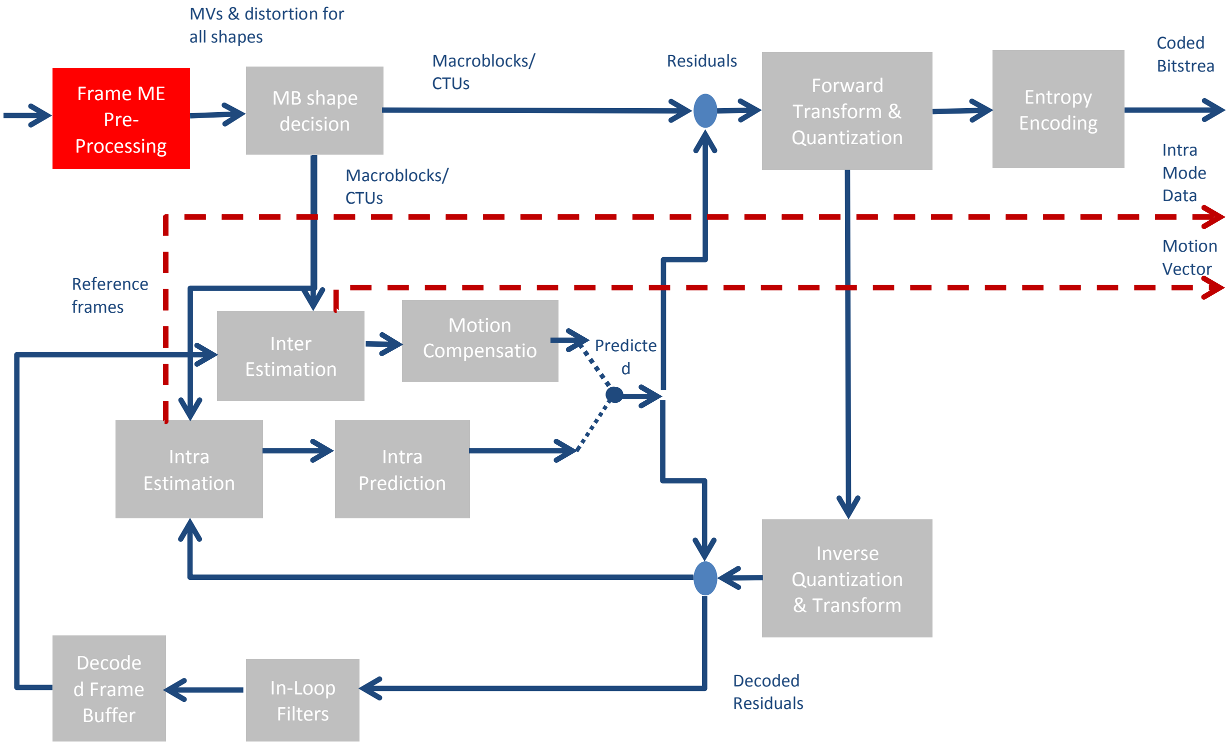


Figure 35: Use of frame-level VME in general encode pipeline

Processing one macroblock per workgroup

The most straightforward OpenCL formulation for this is to set the workgroup size to be equal to (16, 1), thus having one subgroup (with 16 work-items) per workgroup that processes one macroblock using the device-side VME built-in functions. An enqueue with a 2D global size (X, Y) will be used to process all the macroblocks in the image, where X is the pixel width of the image aligned to be in multiples of 16, and Y is the height of the image in units of the macroblock height (16).

However this basic formulation is not recommended for the following reason when there are no macroblock dependencies involved. Referring to Figure 1, there is only one VME unit while there are 8

EUs and 56 hardware threads per subslice for the GEN8 and GEN9 architectures. Thus the VME unit will become the bottleneck and most of the threads will be blocked waiting for the results back from the VME unit. We essentially only require a subset of the available hardware threads to keep the VME units busy, and it is better to generate just enough threads in order to avoid the extra thread launch overhead.

Processing a column of macroblocks per workgroup

A simple approach to minimize the thread launch overhead while keeping the VME units busy is to have each workgroup process all MBs in a image column. This works well for SD images and larger. The workgroup size is still kept as (16, 1), with the global size as (X, 1), where X is the pixel width of the image aligned to be in multiples of 16. The kernel iterates over and processes Y macroblocks, where Y is the height of the image in units of the macroblock height. Figure 3 shows the example of kernel side code using this approach. An example for the host-side code for this approach is shown below.

```
void motion_estimation_kernel(...)
{
    ...
    int mbImageWidth = (width + 15)/ 16;
    int mbImageHeight = (height + 15)/ 16;
    ...
    kernel.setArg(argIndex++, src_img);
    kernel.setArg(argIndex++, ref_img);
    kernel.setArg(argIndex++, sizeof(int), &mbImageHeight);
    ...

    queue.enqueueNDRangeKernel(
        kernel, cl::NullRange,
        cl::NDRange(mbImageWidth * 16, 1, 1),
        cl::NDRange(16, 1, 1), NULL, &evt);
    ...
}
```

Figure 36: OpenCL host enqueue for frame-level VME with no macroblock dependencies

VME with macroblock dependencies (wavefront parallelism)

Motion vectors from neighboring macroblocks are often correlated, and thus using the motion vectors from neighboring macroblocks as reference search window offsets for motion estimation generally yields in a better match. This helps with achieving a higher density encoding because of the lesser magnitude distortions to encode. Moreover motion vectors for macroblocks are differentially encoded w.r.t. the predicted motion vector which is a linear computation based on the motion vectors from its neighboring blocks; and thus biasing the motion vectors to be closer to the predicted motion vector helps with minimizing the bits to encode the motion vectors. The following figure shows the macroblock dependencies for an AVC progressive frame.

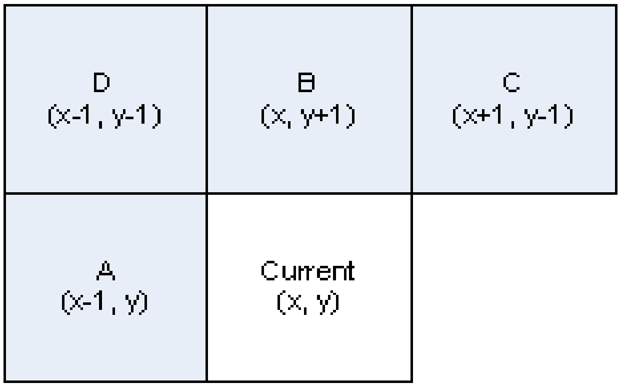


Figure 37: AVC macroblock dependencies (26 degree wavefront)

Figure 38 shows the use of VME in the encode loop processing macroblocks handing neighboring dependencies (in red) in a generalized encode pipeline.

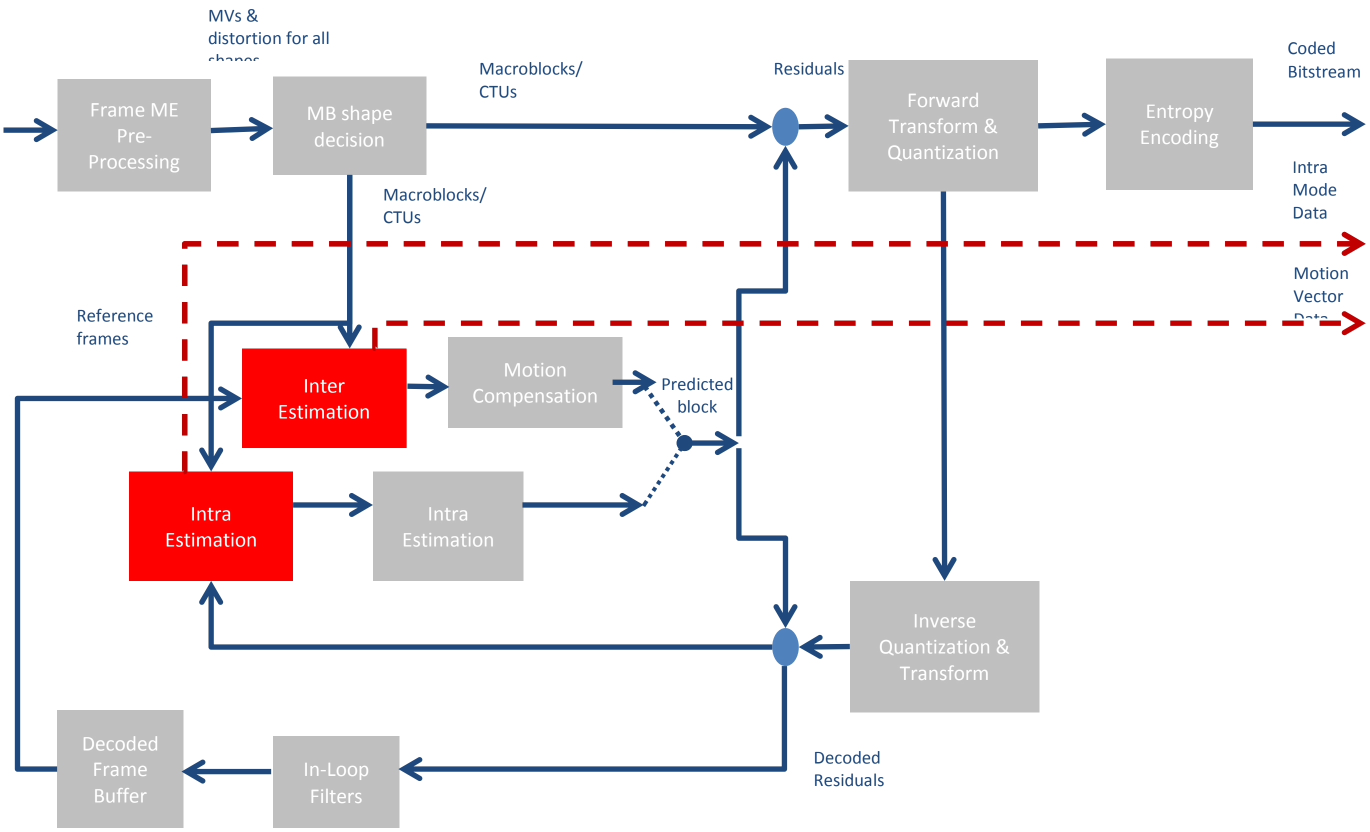


Figure 38: VME with macro-block dependencies in general encode pipeline

When using the OpenCL VME subgroup functions, the subgroups size is fixed as 16, which means than the GEN hardware threads are SIMD16 threads processing 16 work-items in a subgroup. The following discussion will assume that the work-group size is also 16, which means there is one subgroup per work-group, and each subgroup/workgroup processes one macroblock.

In order to handle these macroblock dependencies in the OpenCL programming model, we need to set the workgroups up such that the workgroups processing the dependent macroblocks are guaranteed to get dispatched prior to the dispatch of the depending macroblock while retaining as much parallel processing of macroblocks as possible, and that there is a mechanism in place so that the depending macroblocks are guaranteed to block until the dependent macroblocks complete their processing. The following sections explain the addressing of the above two requirements as efficiently as possible.

Wavefront processing of macroblocks

The wavefront parallel processing paradigm can be used to efficiently handle the order in which macroblocks need to be processed in the encode loop. The efficiency aspect of it comes from maximizing the amount of parallel processing of blocks considering the partial order of processing of blocks created by the macroblock encode dependencies. Figure 39 illustrates the processing order used for a 45-wavefront. Using such a dispatch we can handle dependencies A, B and D for the current macroblock shown in Figure 37, while at the same time keeping as many independent workgroups (each processing one MB) in flight.

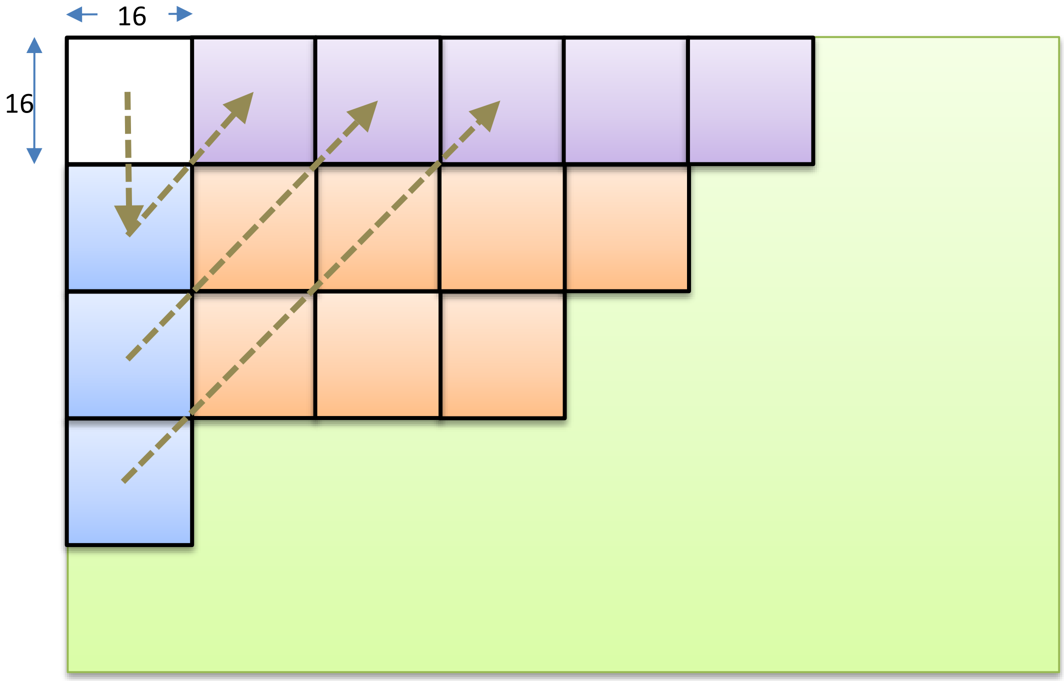


Figure 39: 45 degree wavefront processing

The Intel OpenCL runtime essentially uses a raster dispatch for workgroups. If we are processing a QCIF image as shown below in Figure 40, where each workgroup processes a MB and each MB is assigned a 2D workgroup ID (X, Y), the workgroups are dispatched in the order (0,0), (1,0), …, (10,0), (0,1), (2,0), …, (9,1), (10,1), (0,7),(1,7),…,(9,7),(10,8), (0,8),(1,8),…,(9,8),(10,8).

Note: Using a basic raster order, the AVC dependency order could be satisfied, but it would seriously limit the amount of threads (macroblocks) that could be processed in parallel causing the VME units to be underutilized.

(0,0)	(1,0)		(9,0)	(10,0)
(0,1)	(2,0)		(9,1)	(10,1)
(0,7)	(1,7)		(9,7)	(10,8)
(0,8)	(1,8)		(9,8)	(10,8)

QCIF frame (176x144)		**MBs**	**:** 11x9
Global size: (11*16,9)		**LocalSize**	**:** (16,1)
WG IDs	**:** (X,Y)	**Subgroups/WG :** 1	

Figure 40: Workgroup assignments for frame

If we simply use the 2D workgroup identifiers as the 2D MB identifiers for the image, we will end up processing the MBs in a raster order rather than the desired wavefront order. In order to address this, we need to remap the 2D workgroup identifiers into a different set of 2D MB identifiers, such that the MBs get processed in a wavefront order, even though the workgroups get dispatched in a raster order. This is accomplished by passing in a “launch” buffer that is initialized with the remapped identifiers in the host application into the VME kernel, which the kernel uses to remap the raster order workgroup identifier into the wavefront order MB identifier. The initialization of a 45 degree wavefront launch buffer on the OpenCL host-side is shown below.

```
// Initialize a 45 degree wavefront launch pattern – remaps the 2D raster order
// into a 2D wavefront order.
void get_45_wavefront_launch (cl_short *launch, size_t width, size_t height)
{
   int X = 0, Y = 0, id = 0;
   for (int i = 0; i < width + height - 1; i++) {
      for (int x = X, y = Y; x >= 0 && y < height; x--, y++) {
         launch[id * 2] = x;
         launch[id * 2 + 1] = y;
         id++;
      }
      if (X < (width - 1)) X++;
      else Y++;
   }
}

...
void motion_estimation_kernel(...)
{
    ...
    // The launch buffer is used as an input argument buffer by the VME
    // kernel that processes macroblocks with neighbor block dependencies.
    cl_short *launchMem = new cl_short[ mbImageWidth * mbImageHeight * 2 ];
    get_45_wavefront_launch( launchMem, mbImageWidth, mbImageHeight );
    cl::Buffer launchBuffer(
        context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
        mbImageWidth * mbImageHeight * sizeof(cl_short2), launchMem, NULL);
    ...
    kernel.setArg(argIndex++, launchBuffer);
    ...
    int mbImageWidth = (width + 15)/ 16;
    int mbImageHeight = (height + 15)/ 16;
    queue.enqueueNDRangeKernel(
        kernel, cl::NullRange,
        cl::NDRange(mbImageWidth  * 16, mbImageHeight, 1),
        cl::NDRange(16, 1, 1), NULL, &evt);
    ...
}
```

Example 34: 45 degree wavefront launch buffer initialization and use in OpenCL host

The initialization of the launch buffer need to performed only once and is pre-processing step.

On the OpenCL kernel side, the workgroup identifiers need to be remapped into the wavefront identifiers using the pre-computed launch buffer as shown below.

```
__kernel __attribute__((reqd_work_group_size(16,1,1)))
void block_motion_estimate_intel(
    __read_only image2d_t   src_img,
    __read_only image2d_t   ref_img,
    __global short2*        motion_vector_buffer,
    ...
    __global short2*       launch_buffer)
{
    int2 gid = { get_group_id(0), get_group_id(1) };
    int2 ngrp = { get_num_groups(0), get_num_groups(1) };

    int2 mbid;
    int lid = gid.y * ngrp.x + gid.x;
    mbid.x = launch_buffer[lid].x;
    mbid.y = launch_buffer[lid].y;

    srcCoord.x = mbid.x * 16;
    srcCoord.y = mbid.y * 16;

    ...
}
```

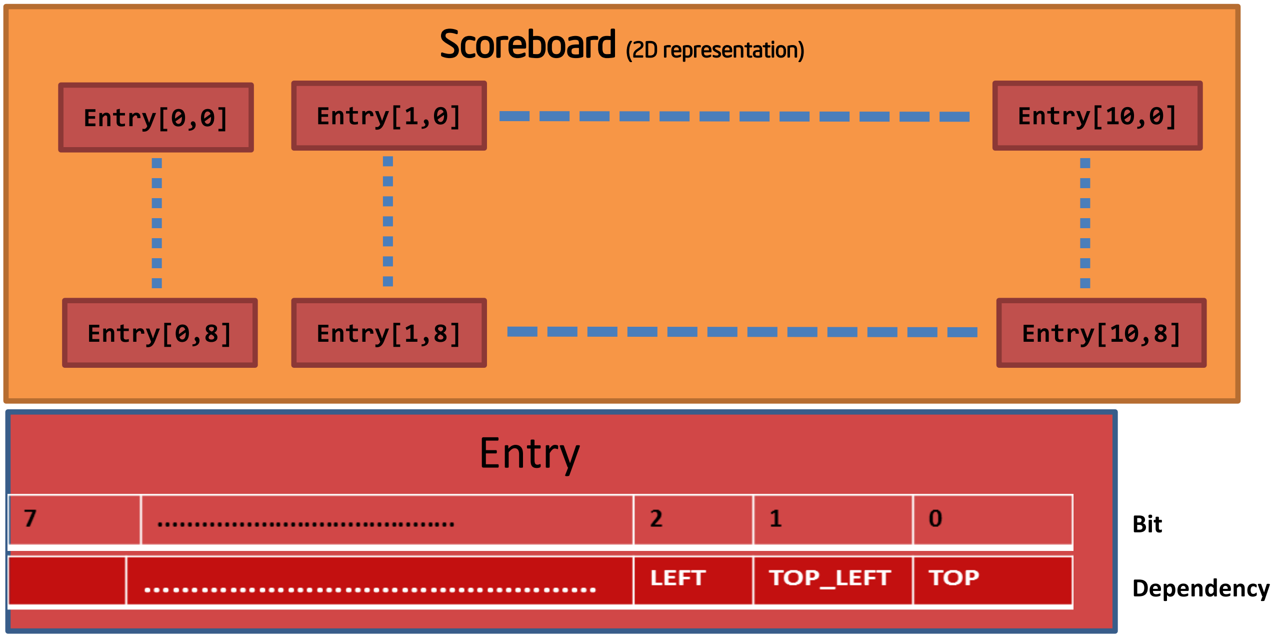
Example 35: Processing macroblocks using wavefront identifiers in OpenCL kernel

Software Scoreboarding

Software scoreboarding is the mechanism to satisfy the requirement that the depending macroblocks need to block until the dependent macroblocks complete their processing.

Representation

A scoreboard is a buffer that contains a table of scoreboard entries that span the linearized macroblock 2D identifier space. Each entry in the table contains the real time status of dependencies of a macroblock. Each entry has the form of a bit field array, where the number of bit fields is equal or greater than the maximum number of dependencies that a macroblock needs to handle. Considering the basic AVC style dependencies from Figure 37 for inter-estimation where each macroblock needs to handle 3 dependencies LEFT (A), TOP (B) and TOP_LEFT (D), an 8 bit integer can be used for a scoreboard entry. In the implementation the scoreboard is a 1D buffer with its logical 2D indexes linearized. Note that the scoreboard table is indexed using the logical 2D macroblock index returned from the launch buffer (and not the workgroup index). The following figure illustrates the 2D representation of the scoreboard buffer and its entries.



Bit	7	....	2	1	0
Dependency		....	LEFT	TOP_LEFT	TOP

Figure 41: Scoreboard representation

Initialization phase

A scoreboard initialization phase is required prior to processing each image. In Figure 38, the initialization can be performed in the “Frame ME Pre-Processing” stage that is called for each source image prior to the per macroblock encode loop operations.

The initialization phase will take care of the image boundary cases where there are no or limited dependencies. There will be at least one scoreboard entry with no dependencies. The initializations are performed using an enqueue of a pre-processing OpenCL kernel that executes prior to the enqueue of the VME kernels processing the macroblocks. The additional compute required for the initialization is small as compared to the VME kernel. The initialization kernel and the host code for invoking are shown below.

```
__kernel
void initialize_scoreboard(__global uint* status, int num_mb_x )
{
    int x = get_global_id(0);
    int y = get_global_id(1);
    if( x < num_mb_x )
    {
       int lid = y * width + x;
       if(y == 0)
          status[lid] |= TOP_DEP;
       if(x == 0)
          status[lid] |= LEFT_DEP;
       if((x == 0) || (y == 0))
         status[lid] |= TOP_LEFT_DEP;
       else
         status[lid] = 0;
    }
}
```

Example 36: Scoreboard initialization kernel-side

```
...

{
    ...
    int mbImageWidth = (width + 15)/ 16;
    int mbImageHeight = (height + 15)/ 16;

    cl::Buffer initBuffer(
        context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
        mbImageWidth * mbImageHeight * sizeof(cl_int), status, NULL);

    // Reset the scoreboard buffer
    scoreboard.setArg(0, initBuffer);
    scoreboard.setArg(1, sizeof(int), &mbImageWidth);
    ...
    queue.enqueueNDRangeKernel(
        kernel, cl::NullRange,
        cl::NDRange(mbImageWidth, mbImageHeight, 1),
        cl::NullRange, NULL, &evt);
    ...
}
```

Example 37: Scoreboard initialization host-side

Blocking phase

Each workgroup processing a macroblock continues to poll its scoreboard entry using its macroblock identifier in a loop until all its dependencies have been cleared. Atomic operation with acquire semantics are used to poll in order to synchronize with the atomic operations in the signaling phase. The memory polling will moderately impact the available memory bandwidth. In the Icelake architecture a hardware based thread synchronization monitor is added to reduce the polling frequency. The following example demonstrates the polling function.

```
void poll_scoreboard(int2 mbid, int2 ngrp, __global atomic_int* scoreboard)
{
       int ALL_DEPS = TOP | TOP_LEFT | LEFT;
       int entry = 0;
       while (entry!= ALL_DEPS)
       {
           s =
              atomic_load_explicit(
                     scoreboard + mbid.y * ngrp.x + mbid.x,
                     memory_order_acquire,
                     memory_scope_device);
       }
}
```

Example 38: Scoreboard blocking

Signaling phase

When a workgroup processing a macroblock computes its results which its depending MBs are waiting on, it clears its dependency in the corresponding bit field in the scoreboard entries of its depending MBs. Atomic operations with acquire-release semantics are used to update the scoreboard entry bit fields as multiple workgroups could be clearing their respective dependencies for the same depending MB entry.

```
void signal_scoreboard(int2 mbid, int2 ngrp,__global atomic_int* scoreboard)
{
    if (mbid.y < ngrp.y - 1) {
        atomic_fetch_or_explicit(
                     scoreboard + (mbid.y+1) * ngrp.x + mbid.x,
                     TOP_DEP,
                     memory_order_acq_rel,
                     memory_scope_device );
    }
    if (mbid.x < ngrp.x - 1) {
        atomic_fetch_or_explicit(
                     scoreboard + mbid.y * ngrp.x + mbid.x + 1,
                     LEFT_DEP,
                     memory_order_acq_rel,
                     memory_scope_device );
    }
    if ((mbid.x < ngrp.x - 1) && (mbid.y < ngrp.y - 1)) {
        atomic_fetch_or_explicit(
                     scoreboard + (mbid.y + 1) * ngrp.x + mbid.x + 1,
                     TOP_LEFT_DEP,
                     memory_order_acq_rel,
                     memory_scope_device );
    }
}
```

Example 39: Scoreboard signaling

Processing phase

The kernel processing for a macroblock starts when the blocking phase on the macroblock's scoreboard entry competes. The kernel processing will use the VME computation results from its depending macroblocks. After completing its computation, it enters its signaling phase to update the scoreboard entries of its dependent macroblocks to indicate its completion. The acquire-release semantics of the atomic operations used for the poll and signal operation will ensure that the writes are visible to other threads prior to the scoreboard updates becoming visible.

```
__kernel __attribute__((reqd_work_group_size(16,1,1)))
void block_motion_estimate_intel(
    __read_only image2d_t   src_img,
    __read_only image2d_t   ref_img,
    __global short2*        motion_vector_buffer,
    ...
    __global atomic_int*    scoreboard,
    __global short2*         launch_buffer)
{
    int2 mbid, gid = { get_group_id(0), get_group_id(1) };
    int2 ngrp = { get_num_groups(0), get_num_groups(1) };
    int lid = gid.y * ngrp.x + gid.x;
    mbid.x = launch_buffer[lid].x; mbid.y = launch_buffer[lid].y;
    ...
    // Blocking phase polling the scoreboard.
    if( sub_group_local_id() == 0 ) {
        poll_scoreboard(mbid, ngrp, scoreboard)
    }
    // Processing phase after scoreboard entry for mbid is cleared.
    // Get motion vectors from dependent neighboring macroblocks
    if (mbid.x != 0 || mbid.y != 0) {
       // No neighboring macroblocks to consider.
    }
    else if (mbid.x == 0) {
        predMV1 = motion_vector_buffer [mbid.x * 16 + (mbid.y - 1) * 16 * ngrp.x ];
    }
    else if (mbid.y == 0) {
        predMV1 = motion_vector_buffer [ (mbid.x - 1) * 16 + mbid.y * 16 * ngrp.x ];
    }
    else {
        predMV1 =
            motion_vector_buffer [ (mbid x - 1) * 16 + (mbid.y - 1) * 16 * ngrp.x ];
        predMV2 = motion_vector_buffer [mbid.x * 16 + (mbid.y - 1) * 16 * ngrp.x ];
        predMV3 = motion_vector_buffer [ (mbid.x - 1) * 16 + mbid.y * 16 * ngrp.x ];
    }

    // Perform VME operation.
    ...
    long mvs = intel_sub_group_avc_ime_get_motion_vectors( result );

    // Write results to global memory.
    int index = (mbid.x * 16 + get_local_id(0) ) + (mbid.y * 16 * ngrp.x);
    int2 bi_mvs = as_int2( mvs );
    motion_vector_buffer [ index ] = as_short2( bi_mvs.s0 );


    // Perform signaling phase.
    if( sub_group_local_id() == 0 ) {
        update_scoreboard(mbid, ngrp, scoreboard);
    }
    ...
}
```

Example 40: VME processing using motion vectors from dependent neighbors

The following figure illustrates the flow of software scoreboarding blocking and signaling phases.

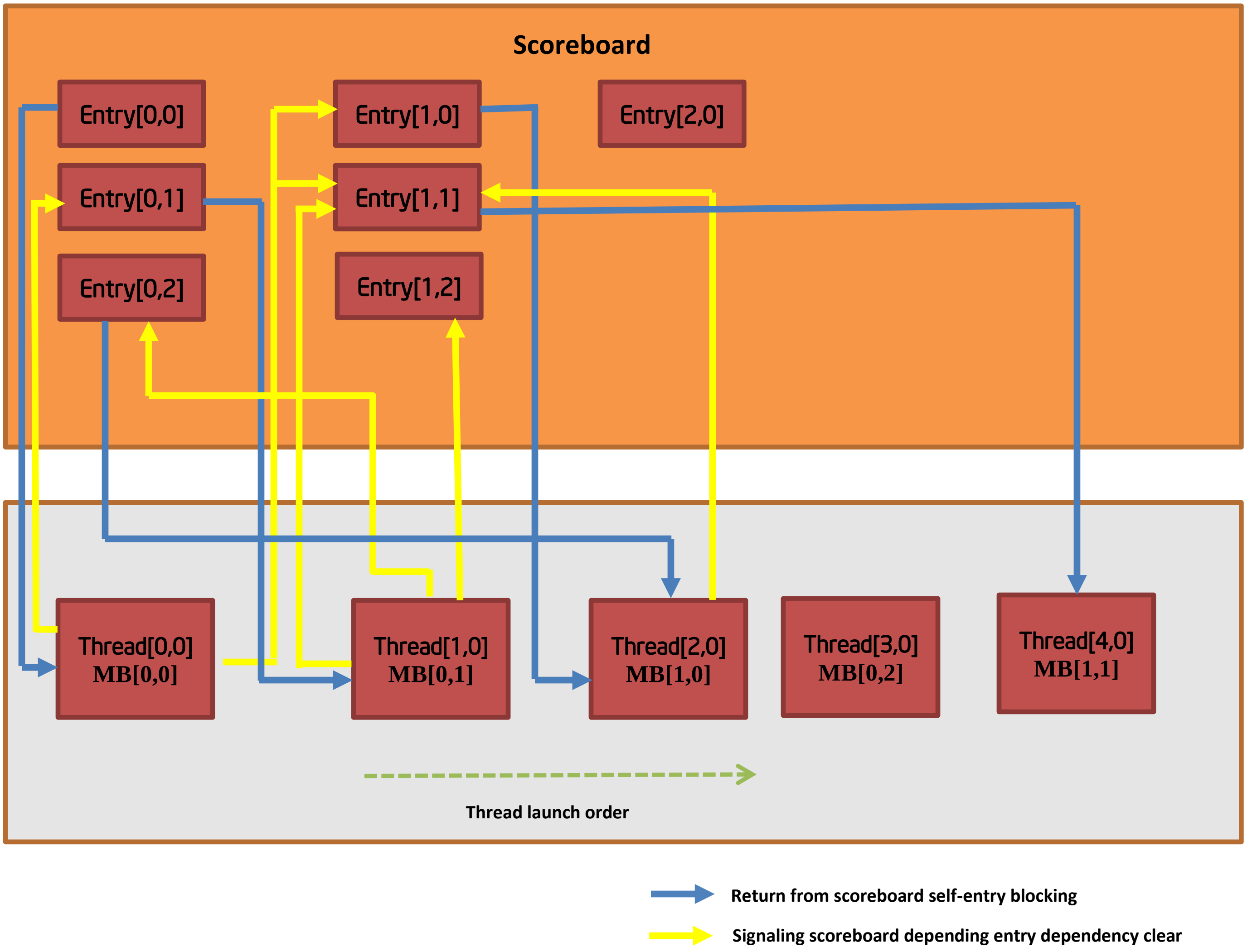


Figure 42: Scoreboard blocking and signaling

In the above figure the initialization phase which runs as a pre-processing phase would have cleared the dependencies for scoreboard entry[0,0], and initializes entry[0,1] and entry[1,0] to having only one dependency to clear, the TOP and LEFT dependencies respectively.

As can be inferred from Figure 42, during the initial and tail end of processing of the image there aren't enough threads in flight and the parallelism is limited, but in between there will be enough threads to keep the VME units busy. As mentioned earlier, since there is only one VME unit while there are 8 EUs and 56 threads per subslice, only a subset of EU threads need to be filled to keep the VME units busy.

Application of OpenCL for HEVC encode pipeline

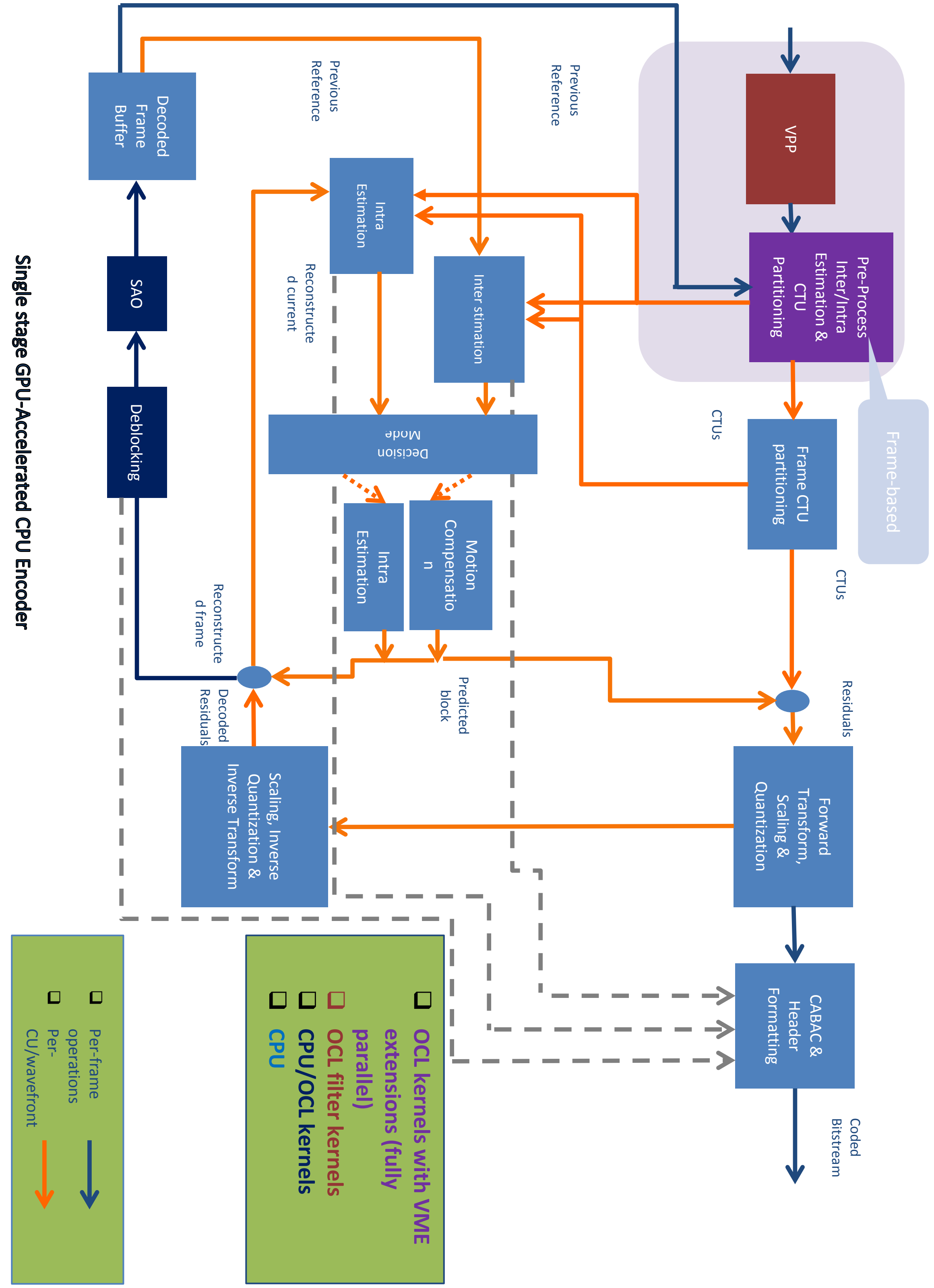


Figure 43: Single stage GPU accelerated CPU encoder

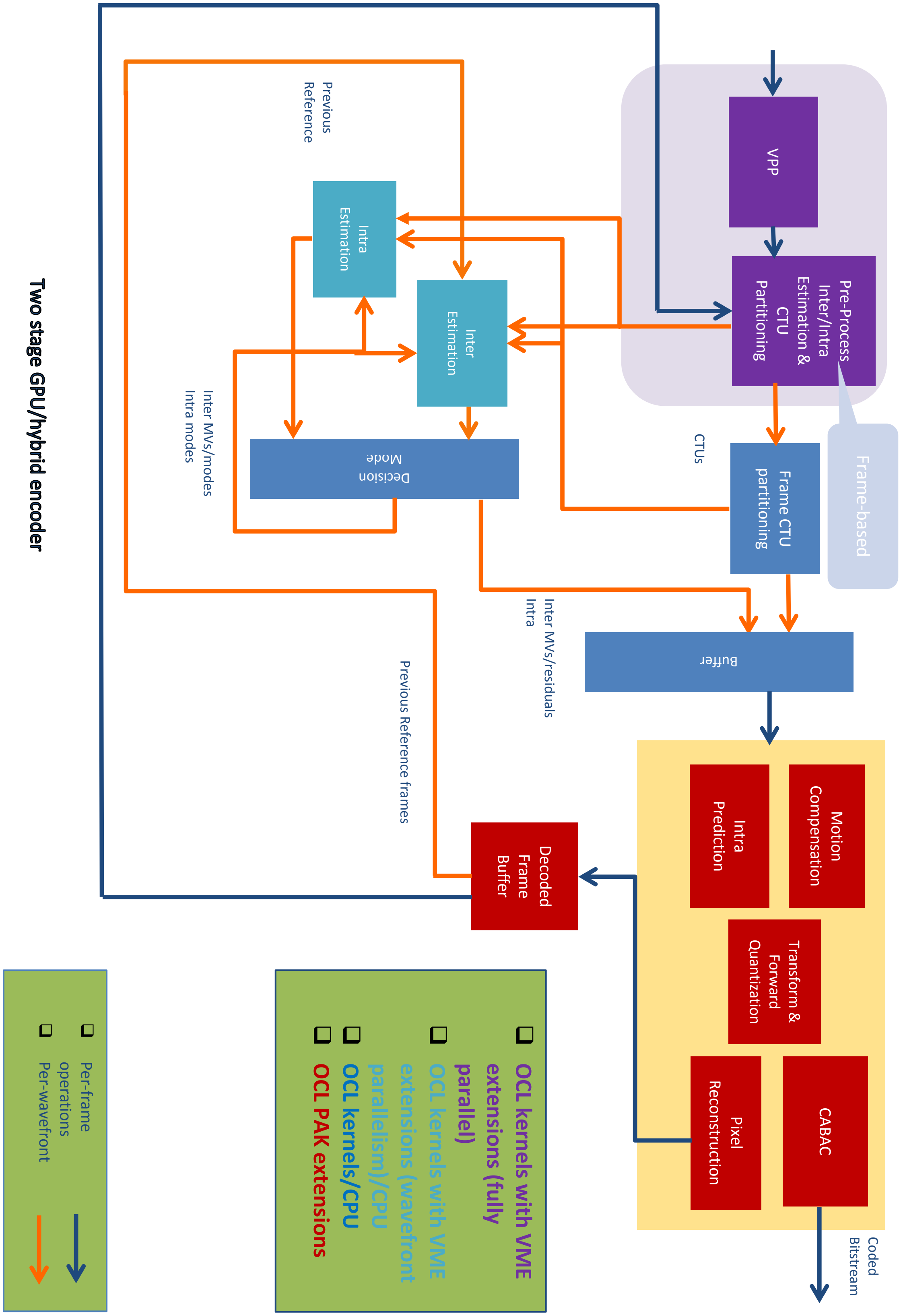


Figure 44: Hybrid two stage GPU encoder

References

1. The Compute Architecture of Intel® Processor Graphics Gen8.
2. The Compute Architecture of Intel® Processor Graphics Gen9.
3. https://www.khronos.org/registry/cl/extensions/intel/cl_intel_subgroups.txt
4. https://www.khronos.org/registry/OpenCL/extensions/intel/cl_intel_device_side_avc_motion_estimation.txt
5. https://www.khronos.org/registry/OpenCL/extensions/intel/cl_intel_media_block_io.txt
6. https://www.khronos.org/registry/OpenCL/extensions/intel/cl_intel_planar_yuv.txt
7. https://www.khronos.org/registry/OpenCL/extensions/intel/cl_intel_va_api_media_sharing.txt