Intro to Advanced Motion Estimation Extension for OpenCL™

This article introduces Intel's advanced motion estimation extension for the OpenCL™ software technology. This extension includes a set of host-callable functions for the frame-based Video Motion Estimation (VME).

This extension builds upon the `cl_intel_motion_estimation` extension by providing block-based estimation and greater control over the estimation algorithm.

The VME extension depends on the OpenCL 1.2 notion of the built-in kernels and on the `cl_intel_accelerator` vendor extension, which provides an abstraction for the specific hardware-accelerated capabilities. This extension reuses the set of host-callable functions and "motion estimation accelerator objects" defined in the `cl_intel_motion_estimation` extension.

This article provides a brief overview of the `cl_intel_accelerator` and `cl_intel_advanced_motion_estimation` extensions. A code example of using these extensions is also included along with an explanation of its results.

For more information on extensions, refer to the `cl_intel_accelerator` and `cl_intel_advanced_motion_estimation` extension descriptions at the Khronos API registry.

Motion Estimation Overview

Motion estimation is the process of determining motion vectors that describe transformation from one 2D image to another, usually from adjacent frames in a video sequence. The motion estimation functions, considered in this article, assume three basic use cases:

- Performing inter-prediction motion estimation on the source and reference images to obtain the best search motion vectors and their associated distortion values.
- Performing skip-checks on the source and reference images by providing a set of motion vectors, and then obtain the corresponding distortion values.
- Performing intra-prediction computations to obtain the best-search prediction modes between adjacent macro-blocks (MBs) and associated residual values.

The functions can be set up to do some or all of these operations in a single enqueue.

The introduced VME functionality exposes part of the hardware acceleration pipeline for video acceleration. The advanced VME extension provides low-level functionality, currently restricted to the single-channel (luma) input images and block matching methods, so motion vectors are computed for rectangular pixel blocks. Chroma channels processing possibility is reserved for intra-prediction computations and is not currently implemented in the code example.

Motion vectors are key elements in the video compression algorithms. Motion vectors are useful for several applications. For example, when generating "slow motion" effects, motion vectors can provide the basis to generate intermediate frames for frame rate (up) conversion. Another example is increasing the original frame rate of the digitized film (24 fps) to match the TV rate.

Motion vectors are also useful for image stabilization: the motion vectors in the entire frame can be averaged to produce a “global” motion vector that can serve as an approximation to a real video camera motion.

The new skip-checks functionality introduced in the advanced VME extension is fundamental in novel video codec algorithms. It provides input for direct encode or skip mode decision maker. Skip mode significantly reduce the number of bits to be coded and amount of calculations in the video codec pipeline as well. For example, encoding video sequences with static scenes or with steady directional camera movement, though static scene can significantly benefit from enabling the skip mode. In such cases multiple macro-blocks of the current frame can be encoded with a minimal number of bits and reconstructed on the decoder side using the corresponding macro-block data from the previous or the next frame without degrading the resulting image quality.

The advanced motion estimation extension consists of the new OpenCL built-in kernel (see section 5.6.1 in the OpenCL 1.2 specification), which performs motion estimation, as well as the accelerator object, which represents the state of the underlying acceleration engine. The kernel is queued for execution from the host using the standard ND-range mechanism.

Both `cl_intel_accelerator` and `cl_intel_advanced_motion_estimation` extensions should be listed in the `CL_DEVICE_EXTENSIONS` string (see Table 4.3 in the OpenCL 1.2 specification) for the Intel® Processor Graphics device in your system. Otherwise you need to update your GPU driver first.

General Accelerator API

Creating an Accelerator Object

Accelerator objects provide a black-box abstraction of software- and/or hardware-accelerated functionality from the OpenCL vendors. Intel `cl_intel_accelerator` vendor extension consists of a unified set of OpenCL runtime APIs, which enable creating, querying, and managing the lifetime of the accelerator objects. The interfaces for this extension are provided in the `cl_ext.h` header.

Just as with the other vendor extension APIs, the `clGetExtensionFunctionAddressForPlatform` function should be used to get pointers to the accelerator APIs:

```
static clCreateAcceleratorINTEL_fn pfn_clCreateAcceleratorINTEL =
(clCreateAcceleratorINTEL_fn)
clGetExtensionFunctionAddressForPlatform(intel_platform_id,
"clCreateAcceleratorINTEL");
```

`clCreateAcceleratorINTEL_fn` is defined as an appropriate function pointer in the `cl_ext.h`.

Accelerator object instances are referenced with the generic `cl_accelerator_intel` type. Notice that every accelerator is always associated with a specific acceleration engine type, which is requested by the application at accelerator object creation time. In the example below, the accelerator type is `CL_ACCELERATOR_TYPE_MOTION_ESTIMATION_INTEL`. Also, descriptors are used to request acceleration engine-specific properties:

```
cl_motion_estimation_desc_intel desc = {
CL_ME_MB_TYPE_16x16_INTEL,
CL_ME_SUBPIXEL_MODE_INTEGER_INTEL,
CL_ME_SAD_ADJUST_MODE_NONE_INTEL,
CL_ME_SEARCH_PATH_RADIUS_16_12_INTEL
};
cl_accelerator_intel accelerator = pfn_clCreateAcceleratorINTEL(context,
CL_ACCELERATOR_TYPE_MOTION_ESTIMATION_INTEL,
```

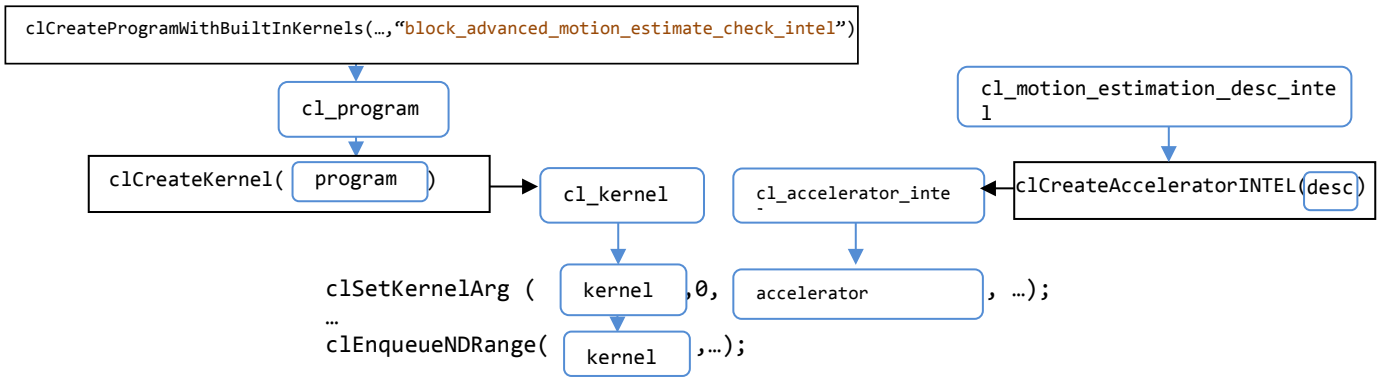
```
        sizeof(cl_motion_estimation_desc_intel), &desc, &err);
```

Refer to the full motion estimation extension specification for the descriptor details. Make sure to handle potential failure of the creation routine when `clCreateAcceleratorINTEL` returns zero for the accelerator handle value. Possible reasons for accelerator creation failure are invalid descriptors or an invalid combination of descriptor values. The extension specification lists all possible error codes and causes.

`clReleaseAcceleratorINTEL` is a complement to the creation API described above. Refer to the Full Frame Motion Estimation Code Example section of this article for the example code.

Using the Accelerator Object

An application can run the accelerated motion estimation functions on an OpenCL device by enqueuing one of the proposed built-in kernels (below). The kernels are enqueued for execution by the regular `clEnqueueNDRangeKernel` OpenCL routine. In turn, a motion estimation accelerator encapsulates the internal state of the motion estimation engine and serves as the kernel argument to the motion estimation built-in kernel. The relationships between the entities are outlined in the following diagram:



Motion Estimation API

Notion of Built-in Kernels

Section 5.6.1 of the OpenCL 1.2 specification introduces the notion of built-in kernels. Specifically, `clCreateProgramWithBuiltInKernels` creates a program object given a context and loads the information related to the built-in kernels into the program object. Notice that the developer does not provide program source code for built-in kernels.

```
cl_program program =
clCreateProgramWithBuiltInKernels(context,1,device,"block_advanced_motion_estimate
_check_intel",&err);
```

The specific built-in kernels are created from the resulting program object:

```
cl_kernel kernel = clCreateKernel(program,
"block_advanced_motion_estimate_check_intel", &err);
```

The kernels can be enqueued for execution by the OpenCL runtime using `clEnqueueNDRangeKernel`.

Built-in Kernel for Motion Estimation

The `cl_intel_advanced_motion_estimation` extension introduces a new built-in kernel for motion estimation with the following signature:

```
_kernel void
block_advanced_motion_estimate_check_intel
(
accelerator_intel_t accelerator,
__read_only  image2d_t src_image,
__read_only  image2d_t ref_image,
uint flags,
uint skip_block_type,
uint search_cost_penalty,
uchar search_cost_precision,
 __global short2 *count_motion_vector_buffer,
 __global short2 *predictor_motion_vector_buffer,
 __global short2 *skip_motion_vector_buffer,
 __global short2 *search_motion_vector_buffer,
 __global char *intra_search_predictor_modes,
 __global ushort *search_residuals,
 __global ushort *skip_residuals,
 __global ushort *intra_residuals );
);
```

This kernel computes motion vectors (MVs), residuals for inter- and intra-search, and skip-check residuals by comparing a 2D image source with a 2D reference image, producing a vector field of the motion vectors and residuals. The algorithm searches the best match of each pixel block in the source image by searching an image region in the reference image, centered on the coordinates of that pixel block cost center in the source image with an offset, defined by the up to 8 prediction motion vectors per pixel block. The cost center is defined by the first prediction motion vector.

When enqueuing this kernel, `global_work_size` and `global_work_offset` determine the region of interest for the input frames. The dimension of the output motion vector image is dependent on the size of the region of interest and partitioning mode specified by the accelerator.

`accelerator` should be a valid accelerator object created by `clCreateAcceleratorINTEL`, where the type of the accelerator must be `CL_ACCELERATOR_TYPE_MOTION_ESTIMATION_INTEL`.

`src_image` and the `ref_image` images should represent an 8-bit luminance information. `image_channel_order` and the `image_data_type` of `src_image/ref_image` are restricted as follows:

Channel Order	Src Channel Data Type
`CL_R`	`CL_UNORM_INT8`

Additional formats will be support by future extensions.

`flags` defines any optional modes or behaviors when computing motion estimation, skip-check and/or intra-prediction algorithms. Currently the only supported value is 0, however, the following token is reserved for future support:

Type	Description
CL_ME_CHROMA_INTRA_PREDICT_ENABLED_INTEL	Enabled chroma-based intra-prediction.

Chroma-based operations require the NV12 source and reference images (NV12 requires the NV12 format OpenCL extension).

`skip_block_type` flag specifies the sub-block size used during evaluating skip checks. The specified sub-block size will determine the data layout of the `skip_motion_vector_buffer` array:

Type	Sub-block size	MVs per MB entry
CL_ME_MB_TYPE_16x16_INTEL	16x16	1
CL_ME_MB_TYPE_8x8_INTEL	8x8	4

This flag is relevant only for skip-checks.

A cost function scheme can be specified for motion search. Distortion for a MV is computed as a sum of the SAD (Sum of Absolute Differences) and the MV cost penalty. Cost penalty is computed based on the distance between the computed MV and a specific cost-center. This cost-center is specified as the first predictor motion vector configured for a given MB. The `search_cost_penalty` argument specifies the cost penalty function and can be configured for low, normal, or high penalty. The `search_cost_precision` argument is used to configure the range of the cost function by specifying the precision of control points at which the cost penalties are applied to quarter, half, full, or double pixel precision. The cost penalties at in-between control points are linearly interpolated. Generally, a low penalty can be used when using low quantization parameter values during encoding and a high penalty - when using high quantization parameter values.

`search_cost_penalty` defines the cost function scheme used in computing cost penalties.

Type	Description
CL_ME_COST_PENALTY_NONE_INTEL	penalty is zero
CL_ME_COST_PENALTY_LOW_INTEL	penalty for low motion

Type	Description
CL_ME_COST_PENALTY_NORMAL_INTEL	penalty for normal motion
CL_ME_COST_PENALTY_HIGH_INTEL	penalty for high motion

search_cost_precision defines the pixel precision of the cost penalty calculations. If the search_cost_penalty flag is set to CL_ME_COST_PENALTY_NONE_INTEL, this argument is ignored. Possible values are:

Type	Description
CL_ME_COST_PRECISION_QPEL_INTEL	quarter pixel
CL_ME_COST_PRECISION_HPEL_INTEL	half pixel
CL_ME_COST_PRECISION_PEL_INTEL	full pixel
CL_ME_COST_PRECISION_DPEL_INTEL	double pixel

count_motion_vector_buffer defines the number of predictor motion vectors and skip-check motion vectors defined for each macro-block (MB). The buffer contains an array of short integer pairs, one pair per MB. The indices of the array correspond to the contiguous row-major block layout of the input frame. The first value in each pair defines the number of predictor motion vectors for a given MB; this value defines the range of valid entries for the MB contained within the predictor_motion_vector_buffer array. The second value in each pair defines the number of skip-check motion vectors for the MB; this value defines the range of valid entries in the skip_motion_vector_buffer array. All size values must be between 0 and 8 inclusive.

predictor_motion_vector_buffer defines an input array of signed short integer predictor MVs with the quarter-pixel resolution. The array is partitioned into clusters of 8 motion vectors per MB in contiguous row-major ordering. The buffer layout assumes the maximum size of 8 predictor MVs per MB even if the count_motion_vector_buffer array specifies a smaller predictor count. If the value of the search_cost_penalty argument does not equal CL_ME_COST_PENALTY_NONE_INTEL, the first predictor MV for each MB is used as the cost center for cost penalty calculations. If the array, passed to the count_motion_vector_buffer argument, specifies a predictor size of zero for all macro-blocks this argument can be NULL. Please

notice that no MVs search is performed if the predictor size is zero and `predictor_motion_vector_buffer` is NULL. Anyway, the `block_advanced_motion_estimate_check_intel` built-in kernel can be used for skip checking in this case. Please notice difference with the `cl_intel_motion_estimation` extension and the `predictor_motion_vector_buffer` argument treatment there. In the `cl_intel_motion_estimation` extension MVs search performed even if `predictor_motion_vector_buffer` is `NULL`. Prediction motion vector is (0,0) (centered on the coordinates of MBs) by default in this case. The `cl_intel_advanced_motion_estimation` extension provides opportunity of up to 8 prediction MVs setup per MB. Full frame motion estimation code example demonstrates this.

`skip_motion_vector_buffer` defines an input array of signed short integer skip-check MVs. The array is partitioned into clusters of 8 sets of motion vectors per MB, in contiguous row-major ordering. The value of `skip_block_type` determines the number of MVs for each of the 8 entries.

The buffer layout assumes the maximum size of 8 MV entries per MB, even if the `count_motion_vector_buffer` array specifies a smaller skip-check count. If the array passed to `count_motion_vector_buffer` specifies a skip-check size of zero for all macro blocks, no skip-check computation is performed and this argument can be NULL. Application needs to provide `NULL` as the `arg_value argument` to `clSetKernelArg()`in this case.

`search_motion_vector_buffer` defines an output array of signed short integers pairs defining the best search motion vectors per macro block. The array contains 1, 4, or 16 motion vectors per MB in contiguous row-major ordering. The number of vectors per MB is determined by the value of `mb_block_type` specified during the creation of the accelerator object:

Value of mb_block_type	Sub-block size	MVs per MB entry
`CL_ME_MB_TYPE_16x16_INTEL`	16x16	1
`CL_ME_MB_TYPE_8x8_INTEL`	8x8	4
`CL_ME_MB_TYPE_4x4_INTEL`	4x4	16

`intra_search_prediction_modes_buffer` specifies an output buffer containing a sequence of signed chars describing the predictor modes used during motion estimation. The array is divided into a sequence of 22 bytes per MB in contiguous row-major ordering. Each entry in the array has the following form:

```
struct search_predictor_modes
{
       char luma_16x16_block;
       char luma_8x8_block[4];
       char luma_4x4_block[16];
```

```
        char chroma_8x8_block;
};
```

Each value in the `luma_8x8_block` and `luma_4x4_block` arrays contains one of the following constants:

```
CL_ME_LUMA_PREDICTOR_MODE_VERTICAL_INTEL
CL_ME_LUMA_PREDICTOR_MODE_HORIZONTAL_INTEL
CL_ME_LUMA_PREDICTOR_MODE_DC_INTEL
CL_ME_LUMA_PREDICTOR_MODE_DIAGONAL_DOWN_LEFT_INTEL
CL_ME_LUMA_PREDICTOR_MODE_DIAGONAL_DOWN_RIGHT_INTEL
CL_ME_LUMA_PREDICTOR_MODE_VERTICAL_RIGHT_INTEL
CL_ME_LUMA_PREDICTOR_MODE_HORIZONTAL_DOWN_INTEL
CL_ME_LUMA_PREDICTOR_MODE_VERTICAL_LEFT_INTEL
CL_ME_LUMA_PREDICTOR_MODE_HORIZONTAL_UP_INTEL
```

The value of `luma_16x16_block` contains one of the following constants:

`CL_ME_LUMA_PREDICTOR_MODE_VERTICAL_INTEL`

`CL_ME_LUMA_PREDICTOR_MODE_HORIZONTAL_INTEL`

`CL_ME_LUMA_PREDICTOR_MODE_DC_INTEL`

`CL_ME_LUMA_PREDICTOR_MODE_PLANE_INTEL`

The `chroma_8x8_block` only contains valid values if the `CL_ME_CHROMA_INTRA_PREDICT_ENABLED` flag is set. If enabled, the `chroma_8x8_block` contains one of the following constants:

`CL_ME_CHROMA_PREDICTOR_MODE_VERTICAL_INTEL`

`CL_ME_CHROMA_PREDICTOR_MODE_HORIZONTAL_INTEL`

`CL_ME_CHROMA_PREDICTOR_MODE_DC_INTEL`

`CL_ME_CHROMA_PREDICTOR_MODE_PLANE_INTEL`

This argument can be `NULL`.

`search_residuals` defines an output buffer containing vectors of unsigned short SAD-adjusted values corresponding to the best search motion vectors populated in the `search_motion_vector_buffer` array. The array is divided into one vector per MB in contiguous row-major block ordering. Each vector contains 1, 4, or 16 components depending on the value of `mb_block_type` specified during the creation of the accelerator object. This argument can be NULL.

`skip_residuals` defines an output buffer containing vectors of unsigned short SAD-adjusted values corresponding to the skip-check MVs defined by `skip_motion_vector_buffer`. The array is partitioned into clusters of 8 sets of residual values per MB, in contiguous row-major ordering. The value of `skip_block_type` determines the number of values in each of the 8 entries.

The buffer layout assumes the maximum size of 8 residual values per MB, however the number of valid residual entries corresponds to the skip-check MV count specified in `count_motion_vector_buffer` for each MB. This argument can be NULL.

`intra_search_residuals` defines an output buffer of unsigned short SAD-adjusted vectors that correspond to the residual values used during intra-prediction. The buffer contains 4 values per MB in contiguous row-major ordering using the following layout:

```
struct intra_search_residuals
```

```
{
       short luma_16x16_block_residual;
       short luma_8x8_block_residual;
       short luma_4x4_block_residual;
       short chroma_8x8_block_residual;
};
```

The `chroma_8x8_block_residuals` value is only valid if the `CL_ME_CHROMA_INTRA_PREDICT_ENABLED` flag is set. This argument can be `NULL`.

`clEnqueueNDRangeKernel()` for the built-in kernel returns the usual error codes, augmented with a few VME specific error codes, described in the extension specification document. Particularly, notice that this built-in kernel requires the local size to be `NULL` to let the work-group size be determined at runtime, and it requires 2D ND-range. Otherwise the `clEnqueueNDRangeKernel()` call fails and returns an error as described in the specification.

Full Frame Motion Estimation Code Example

The following code snippet demonstrates how to set up and queue a simple full-frame motion estimation pass for 8x8 pixel blocks (and 4 resulting motion vectors per macro-block). This code can be easily extended for skip-checking and intra-frame MVs search functionalities supported by built-in kernel.

```
    cl_platform_id platform;
    cl_context context;
    cl_device_id device;
    cl_command_queue queue;

// Initialize OpenCL via selecting Intel platform, create context with GPU device
and a queue for the device as usual
…

// Get the func pointers to the accelerator routines
static clCreateAcceleratorINTEL_fn pfn_clCreateAcceleratorINTEL =
(clCreateAcceleratorINTEL_fn)
clGetExtensionFunctionAddressForPlatform(platform, "clCreateAcceleratorINTEL");

// Create the program and the built-in kernel for the motion estimation
cl_program program =
clCreateProgramWithBuiltInKernels(context,1,device,"block_advanced_motion_estimate
_check_intel",NULL);
cl_kernel kernel = clCreateKernel(program,
"block_advanced_motion_estimate_check_intel", NULL);

// Create the accelerator for the motion estimation
    cl_motion_estimation_desc_intel desc = { // VME API configuration knobs
// Number of motion vectors per source pixel block, here 4 vectors per block
       CL_ME_MB_TYPE_8x8_INTEL,
       CL_ME_SUBPIXEL_MODE_INTEGER_INTEL, // Motion vector precision
// Adjust mode for the residuals, we don't use them in this tutorial anyway:
       CL_ME_SAD_ADJUST_MODE_NONE_INTEL,
       CL_ME_SEARCH_PATH_RADIUS_16_12_INTEL // Search window radius
    };
    cl_accelerator_intel accelerator =
        pfn_clCreateAcceleratorINTEL(context,
        CL_ACCELERATOR_TYPE_MOTION_ESTIMATION_INTEL,
        sizeof(cl_motion_estimation_desc_intel), &desc, 0);

    // Constants
    const cl_uint kPredictors = 8; // number of predictor MVs
    //Predictor MVs offset in quarter-pixels resolution
    const cl_short kPredictorX0 = 96;
    const cl_short kPredictorY0 = 80;
    const cl_short kPredictorX1 = 0;
```

```
    const cl_short kPredictorY1 = 0;

    // Input images
    cl_image_format format = { CL_R, CL_UNORM_INT8 }; // luminance plane
    cl_mem srcImage = clCreateImage2D(context, CL_MEM_READ_ONLY, &format,
        width, height, 0, pSrcBuf, &err);
    cl_mem refImage = clCreateImage2D(context, CL_MEM_READ_ONLY, &format,
        width, height, 0, pRefBuf, &err);

    // Compute number of output motion vectors
    const int mbSize = 8; // size of the (input) pixel motion block
    size_t widthInMB  = (width + mbSize - 1) / mbSize;
    size_t heightInMB = (height + mbSize - 1) / mbSize;
    // Each Src block has 2x2 MVs if mbSize is 8
    size_t widthInMV  = widthInMB*2;
    size_t heightInMV = heightInMB*2;

    // Output buffer for MB motion vectors
    cl_mem outMVBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
      widthInMV * heightInMV * sizeof(cl_short2), 0, &err);
    // Output buffer for search residuals vectors
    cl_mem residualsBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
      widthInMV * heightInMV * sizeof(cl_ushort), 0, &err);

    //Initialize counters buffer and predictor MVs buffer
    cl_short2 *countMem = new cl_short2[widthInMB  * heightInMB];
    for( int i = 0; i < widthInMB  * heightInMB; i++ )
    {
        countMem[ i ].s[ 0 ] = kPredictors; //[0,8] number of predictor motion
vectors
        countMem[ i ].s[ 1 ] = 0; //for skip motion vectors  - we don’t perform
skip checks in this example
    }

    cl_short2 *predMem = new cl_short2[widthInMB  * heightInMB * 8 ]; //buffer
layout assumes maximal number predictor MVs = 8
    for( int i = 0; i < widthInMB  * heightInMB; i++ )
    {
        //MB center (cost center if cost penalty calculation is enabled)
        for( int j = 0; j < 1; j++ )
        {
            predMem[ i * 8 + j ].s[ 0 ] = 0;
            predMem[ i * 8 + j ].s[ 1 ] = 0;
        }
        //  diagonal up right
        for( int j = 1; j < 2; j++ )
        {
            predMem[ i * 8 + j ].s[ 0 ] = kPredictorX0;
            predMem[ i * 8 + j ].s[ 1 ] = kPredictorY0;
        }
        //  diagonal up left
        for( int j = 2; j < 3; j++ )
        {
            predMem[ i * 8 + j ].s[ 0 ] = -kPredictorX0;
            predMem[ i * 8 + j ].s[ 1 ] = kPredictorY0;
        }
        //  diagonal down right
        for( int j = 3; j < 4; j++ )
        {
            predMem[ i * 8 + j ].s[ 0 ] = kPredictorX0;
            predMem[ i * 8 + j ].s[ 1 ] = -kPredictorY0;
        }
        //  diagonal down left
        for( int j = 4; j < 5; j++ )
        {
            predMem[ i * 8 + j ].s[ 0 ] = -kPredictorX0;
            predMem[ i * 8 + j ].s[ 1 ] = -kPredictorY0;
```

```
        }
        //  Again MB center to reduce initialization code amount; Feel free to
replace with something more useful.
        // For example: right, left, up, down, etc.
        for( int j = 5; j < 8; j++ )
        {
            predMem[ i * 8 + j ].s[ 0 ] = kPredictorX1;
            predMem[ i * 8 + j ].s[ 1 ] = kPredictorY1;
        }
    }
    // Create buffers from initialized memory.
    cl_mem countBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR,
      widthInMB  * heightInMB * sizeof(cl_short2), &countMem, &err);

    cl_mem predBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR,
      widthInMB  * heightInMB * 8 * sizeof(cl_short2), &predMem, &err);


    // Setup parameters for the built-in kernel
    clSetKernelArg(kernel, 0, sizeof(cl_accelerator_intel), &accelerator);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &srcImage);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &refImage);

    unsigned imageType = 0; //Only 0 is supported now
    clSetKernelArg(kernel, 3, sizeof(unsigned), &imageType);

    unsigned skipBlockType = 0; //Don't perform skip-checking in this example.
Will be ignored.
    clSetKernelArg(kernel, 4, sizeof(unsigned), &skipBlockType);

    unsigned costPenalty= CL_ME_COST_PENALTY_HIGH_INTEL; //Cost penalty for high
motion
    clSetKernelArg(kernel, 5, sizeof(unsigned), &costPenalty);

    unsigned costPrecision = CL_ME_COST_PRECISION_HPEL_INTEL; //Cost precision is
half-pixel
    clSetKernelArg(kernel, 6, sizeof(unsigned), &costPrecision);

    clSetKernelArg(kernel, 7, sizeof(cl_mem), &countBuffer);
    clSetKernelArg(kernel, 8, sizeof(cl_mem), &  predBuffer);
    clSetKernelArg(kernel, 9, sizeof(cl_mem), NULL); // no skip checks
    clSetKernelArg(kernel, 10, sizeof(cl_mem), &outMVBuffer); //search result MVs
    clSetKernelArg(kernel, 11, sizeof(cl_mem), NULL); // no intra MVs search
    clSetKernelArg(kernel, 12, sizeof(cl_mem), &residualsBuffer); // search
residuals
    clSetKernelArg(kernel, 13, sizeof(cl_mem), NULL); // no skip residuals
    clSetKernelArg(kernel, 14, sizeof(cl_mem), NULL); // no intra residuals

    // Run the kernel
    // Notice that it *requires* to let runtime determine the local size, and
requires 2D NDRange
    const size_t originROI[2] = { 0, 0 };
    const size_t sizeROI[2] = { width, height};
    clEnqueueNDRangeKernel(queue, kernel, 2, originROI, sizeROI, NULL, 0, 0, 0);

    // Read resulting motion vectors and residuals
    clEnqueueReadBuffer(queue, outMVBuffer, CL_TRUE, 0,
widthInMV  * heightInMV * sizeof(cl_short2), pMVOut, 0, 0, 0);

    clEnqueueReadBuffer(queue, residualsBuffer, CL_TRUE, 0,
widthInMV  * heightInMV * sizeof(cl_short), pResiduals, 0, 0, 0);


    //clReleaseAcceleratorINTEL(accelerator);
    // Release other resources
```

...

Example Results

The pictures below show two frames (reference and source) and computed motion vectors overlaid on the second frame. Specifically, the vectors are rendered as the strokes of the appropriate magnitude. So they point to the new (actually best-matched) positions of the pixel blocks.

Notice the radial pattern of the motion vectors, which is due to the nature of the transformation between frames (zoom in addition to the camera movement).

VME Performance versus Quality Considerations

Pixel block size impact on the image quality and performance is already comprehensively discussed in the `cl_intel_motion_estimation` extension introduction, which can be found at https://software.intel.com/en-us/articles/intro-to-motion-estimation-extension-for-opencl

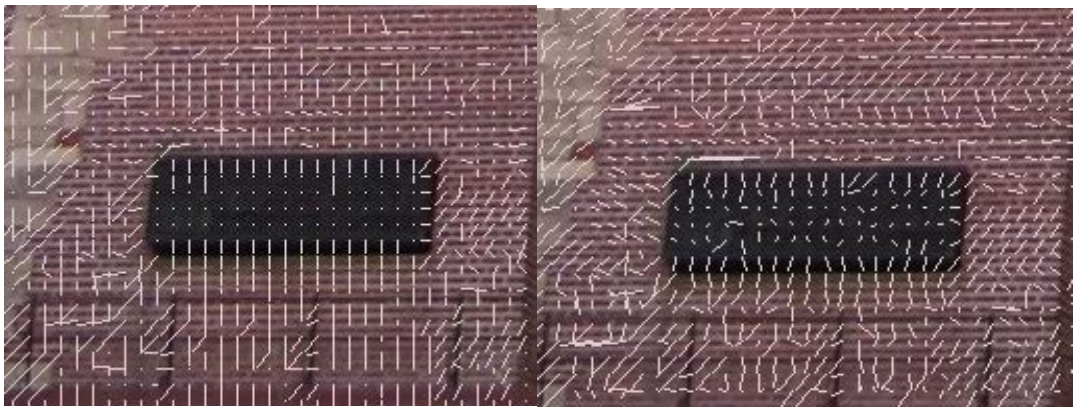
The new advanced motion estimation extension cl_intel_advanced_motion_estimation introduces additional controls, which may significantly impact performance and MVs search result quality and robustness. One of the most important parameters impacting built-in kernel performance is the number of predictor MV's defined by `count_motion_vector_buffer` and `predictor_motion_vector_buffer` kernel arguments. The new advanced extension enables specifying of up to 8 predictor MVs per MB. Additional predictor vectors helps in MVs quality improvement if direction choice is right and closely corresponds to scene objects or camera movement. At the same time computational complexity rise almost linearly with the number of predictor MVs. It is up to user application to choose optimal number of predictor MVs and their direction. The pictures below represent MVs obtained with `cl_intel_motion_estimation` (left) and `cl_intel_advanced_motion_estimation` (right) extension for 8x8 pixel block. This example features 8 predictor vectors in the case of advanced extension. Refer to the code example

for more details. The basic extension used `NULL` as predictor vector built-in kernel argument, which means a single predictor MV targeted to MB default center (0, 0).


As you can see additional predictor MVs available in the advanced extension improves MVs field smoothness.

Cost function penalty and decision parameters may impact the result MVs field quality significantly. Refer to the `search_cost_penalty` and `search_cost_precission` built-in kernel arguments description. This example uses the `CL_ME_COST_PENALTY_HIGH_INTEL` and `CL_ME_COST_PRECISION_HPEL_INTEL` values as the test video sequence contains quite a high motion. Below is example of wrong `search_cost_penalty` argument choice (`CL_ME_COST_PENALTY_NONE_INTEL` or `CL_ME_COST_PENALTY_LOW_INTEL`)

To relaxed cost function precision may also introduce some additional noise in MVs field. See examples for the half-pixel (left) and the double pixel (right) precision below.



In contrast to MV predictors quantity setup, cost function penalty and precision values don’t impact performance significantly.

The following argument setup results in MVs field quality and built-in function performance similar to the one available with the `cl_intel_motion_estimation` extension.

```
// i is the pixel block index
count_motion_vector_buffer[i].s0 = 1; //single MV predictor
count_motion_vector_buffer[i].s1 = 0; //no skip check
predictor_motion_vector_buffer [i].s0 = 0; //X offset
predictor_motion_vector_buffer [i].s1 = 0; //Y offset
search_cost_penalty = CL_ME_COST_PENALTY_NONE_INTEL;
```

Conclusion

Computing motion vectors is a key component of many popular video compression and computer vision algorithms. As it is a computationally-intensive task, pure software implementations might present performance or energy efficiency challenges for some applications. This article introduced an Advanced Video Motion Estimation (Advanced VME) extension for OpenCL™ that leverages hardware-assisted motion vectors estimation. Advanced extension provides additional control to internal VME algorithm parameters setup, which enables significant improvement of the result MVs field quality. The example demonstrates how to employ the set of VME extension host-callable functions for the task of computing motion vectors. Specifically, using this advanced VME extension, you can estimate motion in a frame, while trading off the number of resulting motion vectors and other VME algorithm parameters against computation cost.

About the Authors

Maxim Shevtsov is a Software Architect in the OpenCL performance team at Intel. He received his Masters degree in Computer Science in 2003. Prior to joining Intel in 2005, he was doing various academia studies in computer graphics.

Dmitry Budnikov is a Software Engineer in the OpenCL performance team at Intel. He received his Masters degree in Physics in 1997. Prior to joining Intel in 1998, he was doing various academia studies in digital signal processing.