*Intro to Motion Estimation Extension for OpenCL**

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

This extension depends on the OpenCL 1.2 notion of the built-in kernels and on the `cl_intel_accelerator` vendor extension, which provide an abstraction for the specific hardware-accelerated capabilities.

This article provides a brief overview of the `cl_intel_accelerator` and `cl_intel_motion_estimation` extensions. A code example 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_motion_estimation extension descriptions at the Khronos API registry.

Motion Estimation Overview

Motion estimation is the process of determining motion vectors that describe the transformation from one 2D image to another, usually from adjacent frames in a video sequence. The motion estimation functions, considered in this article, accept full-frame single-channel (luma) images as input, perform a motion search operation, and return a motion vector field as output.

The introduced VME functionality exposes part of the hardware acceleration pipeline for video acceleration. This 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. 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 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_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® HD 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 OpenCL vendors. Intel `cl_intel_accelerator` vendor extension consists of a unified set of OpenCL runtime APIs to create, query, and manage the lifetime of the accelerator objects. The interfaces for this extension are provided in the `cl_ext.h` header. Just as with 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:

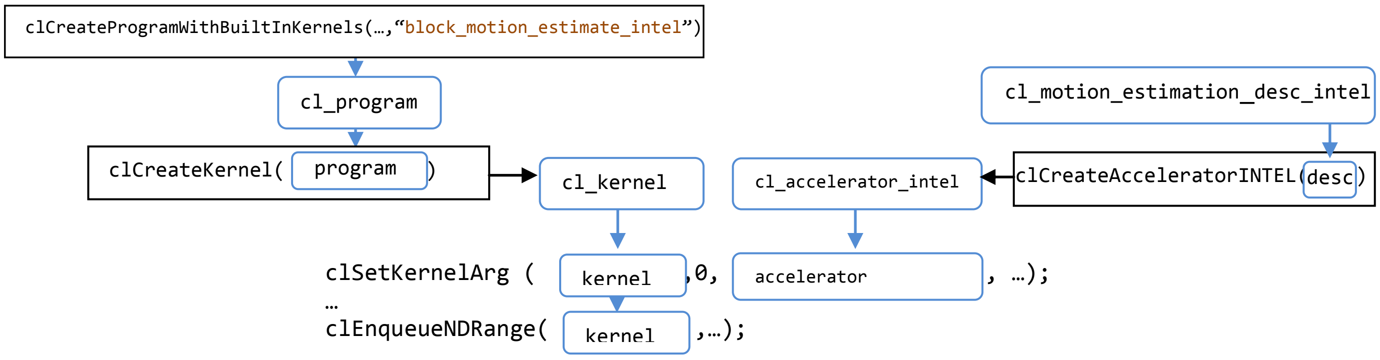
```
cl_motion_estimation_desc_intel desc = {
CL_ME_MB_TYPE_16x16_INTEL,
CL_ME_SUBPIXEL_MODE_INTEGER_INTEL,
CL_ME_SAD_ADJUST_MODE_NONE_INTEL,
CL_ME_SEARCH_PATH_RADIUS_16_12_INTEL
};
cl_accelerator_intel accelerator = pfn_clCreateAcceleratorINTEL(context,
CL_ACCELERATOR_TYPE_MOTION_ESTIMATION_INTEL,
        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 we just discussed. 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:


clCreateProgramWithBuiltInKernels(…,"block_motion_estimate_intel")
cl_program
clCreateKernel(program)
cl_kernel
cl_accelerator_intel
cl_motion_estimation_desc_intel
clCreateAcceleratorINTEL(desc)
clSetKernelArg (kernel ,0, accelerator , …);
…
clEnqueueNDRange(kernel ,…);

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. More 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_motion_estimate_intel",&
err);
```

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

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

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

Built-in kernel for the motion estimation

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

```
_kernel void
block_motion_estimate_intel
(
accelerator_intel_t accelerator,
__read_only  image2d_t src_image,
__read_only  image2d_t ref_image,
__global short2 * prediction_motion_vector_buffer,
__global short2 * motion_vector_buffer,
__global ushort * residuals
);
```

This kernel computes motion vectors by comparing a 2D image source with a 2D reference image, producing a vector field of motion vectors. 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 in the source image (optionally offset by the prediction motion vectors).

When enqueuing this kernel, `global_work_size` and `global_work_offset` determine the region of interest of 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 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

`motion_vector_buffer` represents an output vector field of pixel block motion vectors stored linearly in row-major order. Each entry of the buffer is a motion vector (packed as two 16-bit integer values) for the corresponding pixel block. The buffer needs to be sized appropriately such that it fits the results of all pixel blocks of the source image. The number of returned motion vectors per source pixel block is determined by the `mb_block_type` defined at accelerator creation time. Therefore, the total number of the motion vectors is the number of source (16x16) pixel blocks times the number of returned motion vectors per source block (1, 4, or 16).

This kernel optionally takes a buffer of motion vector predictors via the `prediction_motion_vector_buffer` kernel argument. In many algorithms the motion vectors from the previous frame are used as prediction vectors for the current frame. Prediction vectors can also be used to estimate the motion vectors of the downscaled input image to implement hierarchical motion estimation algorithms. Essentially, using prediction vectors overcomes the hardware limitation on the maximum search radius, as you can offset the neighborhood to be searched, which can be coupled with a multi-pass approach to enable searching within arbitrary areas.

The application can choose not to provide prediction motion vectors by providing `NULL` as the `arg_value` argument to `clSetKernelArg()`, in which case the prediction motion vectors are implied to be (0,0).

A buffer of per-pixel-block distortion values (or “residuals”) can optionally be returned as well, which provides the sum-of-absolute-differences between best-match source and reference frame pixel blocks that produced the corresponding motion vector. The application can choose not to get the residuals by providing `NULL` as the `arg_value` argument to `clSetKernelArg()`, in which case this information is not returned.

Refer to the extension specification document for details.

The `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

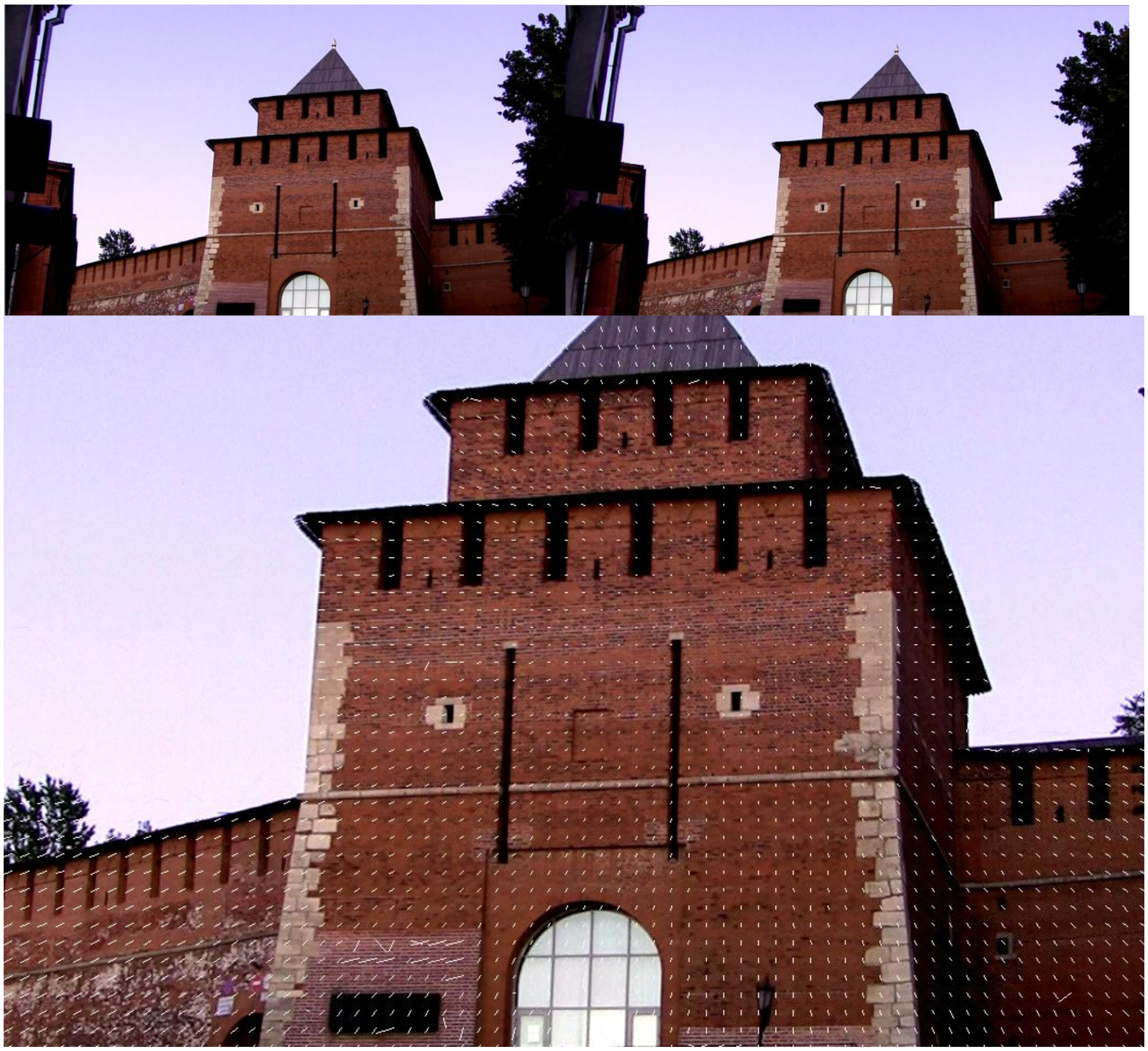
The following code snippet demonstrates how to set up and queue a simple full-frame motion estimation pass for 16x16 pixel blocks (and a single resulting motion vector per block).

```
    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_motion_estimate_intel",N
ULL);
cl_kernel kernel = clCreateKernel(program, "block_motion_estimate_intel", NULL);

// Create the accelerator for the motion estimation
    cl_motion_estimation_desc_intel desc = { // VME API configuration knobs
// Num of motion vectors per source pixel block, here a single vector per block
       CL_ME_MB_TYPE_16x16_INTEL,
       CL_ME_SUBPIXEL_MODE_INTEGER_INTEL, // Motion vector precision
// Adjust mode for the residuals, we don't compute 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);

    // 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 = 16; // size of the (input) pixel motion block
    size_t widthInMB  = (width + mbSize - 1) / mbSize;
    size_t heightInMB = (height + mbSize - 1) / mbSize;
    // Output buffer for MB motion vectors
    cl_mem outMVBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
      widthInMB * heightInMB * sizeof(cl_short2), 0, 0, &err);

    // Setup params 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);
    clSetKernelArg(kernel, 3, sizeof(cl_mem), NULL); // disable predictor motion
vectors
    clSetKernelArg(kernel, 4, sizeof(cl_mem), &outMVBuffer);
    clSetKernelArg(kernel, 5, sizeof(cl_mem), NULL); // disable extra motion block
info output

    // 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
    clEnqueueReadBuffer(queue, outMVBuffer, CL_TRUE, 0,
      widthInMB * heightInMB * sizeof(cl_short2), pMVOut, 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) pixel blocks positions.

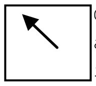
Notice the radial pattern of the motion vectors, which is due to the nature of the transformation between frames (zoom in addition to the camera movement).



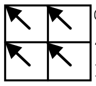
VME Performance versus Quality Considerations

You should carefully consider performance versus quality trade-offs when using hardware-assisted VME through the `cl_intel_motion_estimation` extension. Such trade-offs might be simply a function of the input image size, or of the requested density of motion vectors to be computed. The VME implementation computes motion vectors on a 16x16 source pixel block. You can control the number of output motion vectors to be computed by defining the number of output sub-blocks on each of these 16x16 source pixel blocks. As one would expect, requesting more output motion vectors increases VME computation cost.

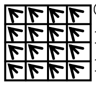
More precisely, the `mb_block_type` field of the `cl_motion_estimation_desc_intel` (refer to the section on the general accelerator API), which is used during VME accelerator creation, defines the number of "sub-blocks" (and hence of motions vectors) within each 16x16 source pixel block:



`CL_ME_MB_TYPE_16x16_INTEL,` a single sub-block per input 16x16 pixel block



`CL_ME_MB_TYPE_8x8_INTEL,` 4 sub-blocks per input 16x16 pixel block



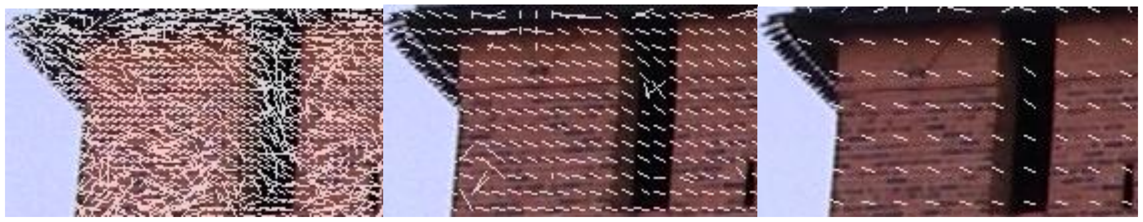
`CL_ME_MB_TYPE_4x4_INTEL,` 16 sub-blocks per input 16x16 pixel block

It is important to understand that sub-blocks are independent. Thus, the smaller the sub-block, the more likely the VME implementation finds a match. Smaller sub-blocks may be appropriate for compression applications that efficiently encode sub-block differences, but less appropriate for feature tracking applications. Using smaller sub-blocks not only increases VME computation expense cost, it also decreases motion vector field smoothness (motion vector directions seem to look more noisy, see the figure below). Feature tracking applications may benefit from a larger sub-block size because they more closely match feature sizes and motion prediction smoothness is often desirable in such applications. Therefore, using larger sub-blocks decreases VME computation expense and improves application performance.

Below are examples of resulting motion vectors fields with the sub-block size varied:

- 4x4 (16 resulting motion vectors per block), the leftmost image
- 8x8 (4 resulting motion vectors per block), the middle image
- 16x16 (single motion vector per block), the rightmost image

Sub-block sizes are specified with the `mb_block_type` field of the `cl_motion_estimation_desc_intel`, a parameter of the VME accelerator routine. Notice the noisiness of the motion vector field in the leftmost image; the noisiness decreases in images to the right:



Conclusion

Computing motions 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. In this article, we introduced a Video Motion Estimation (VME) extension for OpenCL* that leverages hardware-assisted motion vectors estimation. We showed how to employ the set of VME extension host-callable functions for the task of computing motion vectors. Specifically, using this VME extension, one can estimate motion in a frame, while trading off the number of resulting motion vectors against computation cost.

About the Author

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.