Image From Buffer

Sample User's Guide

Intel® SDK for OpenCL™ Applications - Samples

Contents

Legal Information

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

About Image From Buffer

The goal of this sample is to demonstrate how to connect buffer-based kernel and image-based kernel into pipeline using the `cl_khr_image2d_from_buffer` extension. This feature is supported as extension in OpenCL 1.2 and became core functionality in OpenCL 2.0, so any 2.0 device must support it. The functionality enables creating OpenCL image objects, based on OpenCL buffer objects without extra coping, providing dual API to the same piece of memory. Once an image is created, you can use such image features as interpolation and border checking in one kernel, while continuing to access the same physical memory as a regular OpenCL buffer in another kernel.

Main Steps

One way to connect a buffer-based kernel and an image-based kernel is to use `clEnqueueCopyBufferToImage` or `clEnqueueCopyImageToBuffer` between kernels. This approach suffers from extra coping from one physical memory to another. The `cl_khr_image2d_from_buffer` extension enables you to create an image object directly from a buffer object and share the same physical memory. You can do the following to implement such memory sharing:

1. The image-from-buffer functionality may be supported by OpenCL 1.2 devices as extension, so you need to check this support by requesting device info. To do so, get the device extension list and check that the list contains the **`cl_khr_image2d_from_buffer`** substring:

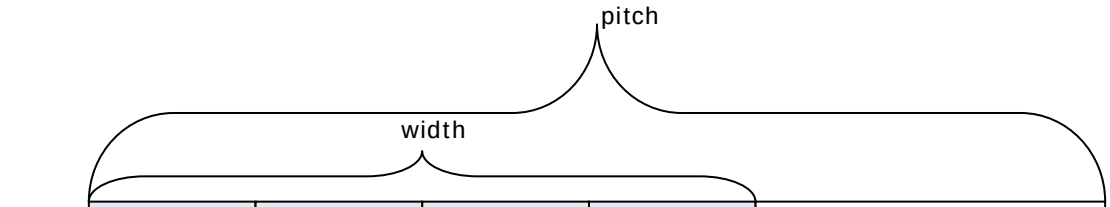
   ```
   err = clGetDeviceInfo(
       device,
       CL_DEVICE_EXTENSIONS,
       extensions_len,
       extensions,
       NULL);
   ```

   The `extensions` is a `char` buffer with null-terminated string that should contain the list of space-separated extension names. For example:

   ```
   “cl_intel_accelerator cl_intel_ctz cl_intel_d3d11_nv12_media_sharing
   cl_intel_dx9_media_sharing cl_intel_motion_estimation cl_khr_3d_image_writes
   cl_khr_byte_addressable_store cl_khr_d3d10_sharing cl_khr_d3d11_sharing
   cl_khr_depth_images cl_khr_dx9_media_sharing cl_khr_gl_depth_images
   cl_khr_gl_event cl_khr_gl_msaa_sharing cl_khr_gl_sharing
   cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics
   cl_khr_icd cl_khr_image2d_from_buffer cl_khr_local_int32_base_atomics
   cl_khr_local_int32_extended_atomics cl_khr_spir”.
   ```

   Note that OpenCL 2.0 has image from buffer as core functionality so for the OpenCL 2.0 device you don’t need to make such check as for OpenCL 1.2.

2. There are two requirements from the specification namely `CL_DEVICE_IMAGE_PITCH_ALIGNMENT` and `CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT` for the image-from-buffer feature. You need to follow these requirements to make your application functional and performant. If you don’t follow the requirements, the image creation might fail.
   The pixels reside in the original buffer line-by-line with some pitch. So the access to pixel with `(x,y)` coordinate is made as `buffer_ptr[y*pitch+x]`.



Line 1	Pixel(1,1)	Pixel(1,2)	Pixel(1,3)	Pixel(1,4)	Padding
Line 2	Pixel(2,1)	Pixel(2,2)	Pixel(2,3)	Pixel(2,4)	Padding
Line 3	Pixel(3,1)	Pixel(3,2)	Pixel(3,3)	Pixel(3,4)	Padding
Line 4	Pixel(4,1)	Pixel(4,2)	Pixel(4,3)	Pixel(4,4)	Padding

The pitch has to be equal or greater than the image width.
All devices have specific requirements for pitch alignment. To obtain the pitch alignment in pixels for a specific device, use the **clGetDeviceInfo** function with the `CL_DEVICE_IMAGE_PITCH_ALIGNMENT` parameter:

```
err = clGetDeviceInfo(
    device,
    CL_DEVICE_IMAGE_PITCH_ALIGNMENT,
    sizeof(cl_uint),
    &pitch_alignment, //pitch alignment in pixels
    NULL);
```

After getting the pitch alignment in pixels, you can calculate the correct pitch size, and (using the **clCreateBuffer** function) create a buffer with size in pixels equal to pitch_size*image_height.

If you create buffer object using `CL_MEM_USE_HOST_PTR` and pointer to the memory, allocated on the host side, then additional restriction to the host pointer is applied. Restrictions are device-specific. To get info on the restrictions, use the `clGetDeviceInfo` function with the `CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT` parameter.

```
err = clGetDeviceInfo(
    device,
    CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT,
    sizeof(cl_uint),
    &ptr_alignment, //pointer alignment in pixels
    NULL);
```

Note that OpenCL 2.0 introduces SVM buffer. SVM is generally not supported for image objects. To create an image object from an SVM buffer, pass the SVM buffer as a pointer to the `clCreateBuffer` function as its `host_ptr` argument with the `CL_MEM_USE_HOST_PTR` flag. In this case `clCreateBuffer` succeeds and returns a valid non-zero buffer object. Then you can use this buffer object for image creation using the image-from-buffer functionality.

An image, created from buffer, uses linear memory representation. It means that pixels are stored in the memory line-by-line. In some cases it may lead to different performance than regular image that usually use tiled storage.

3. Finally, use the **clCreateImage** function to create an image from buffer. The `cl_image_desc.mem_object` field has to be initialized by the created buffer object and `desc.image_row_pitch` has to be initialized by pitch in bytes.

```
cl_image_desc   desc;

desc.image_type = CL_MEM_OBJECT_IMAGE2D;
desc.image_row_pitch = pitch * sizeof(cl_float4);
desc.mem_object = cl_buffer;

cl_intermediate_image = clCreateImage(
    context,
```

```
    CL_MEM_READ_WRITE,
    &format,
    &desc,
    NULL,
    &err);
```

Sample Pipeline

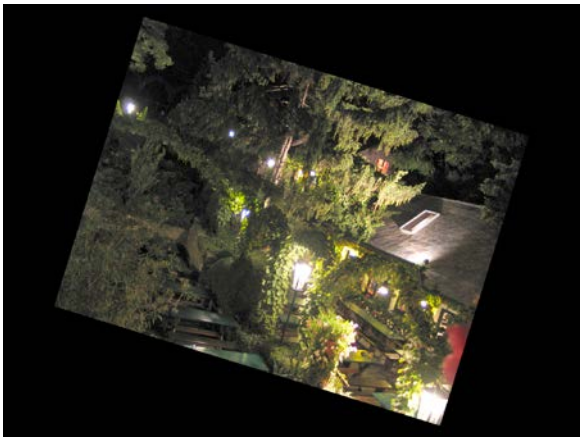
The ImageFromBuffer code sample simulates an image processing pipeline that consists of two kernels. Kernels might be different, but in this specific example the kernels implement **color correction** and **geometry transformation**.

The **color correction kernel** makes simple gamma correction $out = inp^{\frac{1}{4}}$. This operation adds light into dark areas. It works on single pixel without any border condition and interpolation. Therefor such kernel could be efficiently and easy implemented using regular OpenCL buffer.

The **geometry transformation kernel** makes simple geometric transformation using the affine transform. To control the out-of-border access, the kernel should perform the border condition check. Also to make the result smooth, the kernel should perform an interpolation between pixels. For this case the OpenCL image is the best choice because images have built-in support for border and interpolation operations. So the kernel uses image as a source data.

Both the buffer-based and the image-based kernels are connected into one pipeline using the `cl_khr_image2d_from_buffer` extension without extra copying from buffer physical memory to image physical memory.

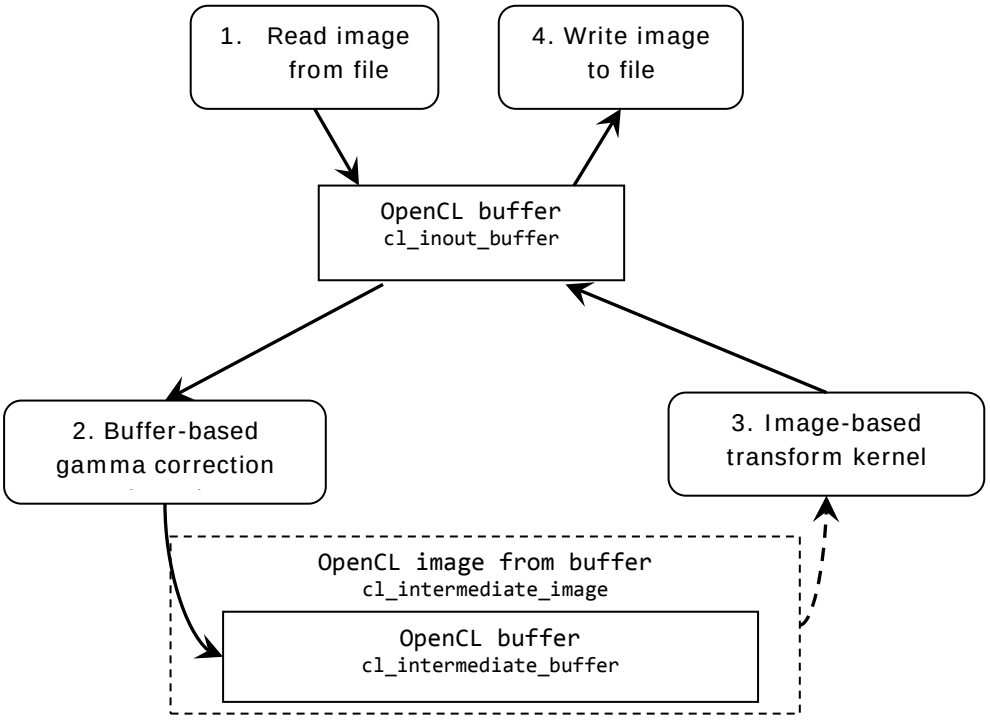
The following pictures represent the estimated processing result: the picture to the left is the source image, while the picture to the right is the result of the image processing pipeline.



The pipeline needs two memory regions for storing:

- Input and output results
- Buffer output for the color correction kernel and at the same time image input for the transformation kernel

The picture below shows data flow and the pipeline steps:



Sample Implementation

The sample code implements the following steps:

1. Makes initial OpenCL initialization, device, context, queue, and kernel creation. These operations reside in the `OpenCLBasic` and `OpenCLProgramMultipleKernels` classes, being regular for any OpenCL application and out of the sample scope.

2. Gets device extension names and checks that the `cl_khr_image2d_from_buffer` extension is supported by the target device. The sample code uses the the `clGetDeviceInfo` function with the `CL_DEVICE_EXTENSIONS` flag. This step is not required for the OpenCL 2.0 device because the image from buffer is core functionality for OpenCL 2.0 and must be supported on all OpenCL 2.0 devices.

3. Gets pitch alignment for buffer to be able to create and use the image from buffer functionality. The code obtains the pitch alignment value using `clGetDeviceInfo` and `CL_DEVICE_IMAGE_PITCH_ALIGNMENT`. The `intermediate_image_pitch` is calculated according to the obtained pitch alignment.

4. Allocates buffers:

    a. Creates `cl_inout_buffer` initialized by the picture pixels.

    b. Allocates OpenCL `cl_intermediate_buffer` using regular `clCreateBuffer` to store intermediate result. The size of the buffer is `intermediate_image_pitch *height*pixelsize`.
    Then OpenCL `cl_intermediate_image` image is created by the `clCreateImage` function based on the `cl_intermediate_buffer`. This image can use the same physical memory as `cl_intermediate_buffer`. So, the code doesn't need to copy data between the buffer and the image.

5. Sets arguments for both kernels and sends them to the command queue for execution using the `clSetKernelArg` and the `clEnqueueNDRangeKernel` functions.

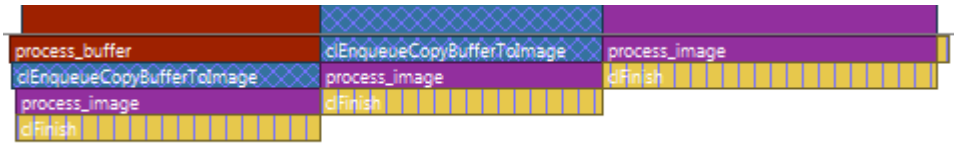
6. The last step is to write the final data into a file.

Understanding OpenCL Performance

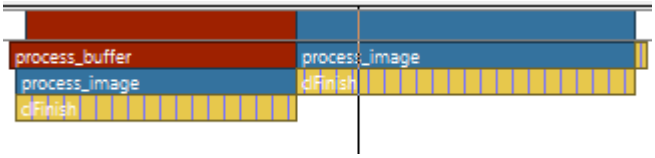
The image-from-buffer feature enables getting better performance for the pipeline with image and buffer processing. If the pipeline is implemented without this feature, then you have to use `clEnqueueCopyBufferToImage` to transfer data from `cl_intermediate_buffer` to `cl_intermediate_image`. As result the total execution time increases.

The pictures below demonstrate the Intel VTune Amplifier XE timelines for different cases:

- The first picture shows the results for pipeline without using the image-from-buffer feature. Additional copying happens between OpenCL kernels, which takes significant amount of time.



- The second picture shows timeline for pipeline improved in case of using the image-from-buffer feature. In this case we save around 1/3 of total pipeline time by removing extra `clEnqueueCopyBufferToImage` operation.



Note, that the image-from-buffer feature gives more benefit for systems with less memory bandwidth than for the systems with higher memory bandwidth. Usually discrete GPU has faster memory then integrated GPU. Therefore, the image-from-buffer feature may provide more benefit for integrated GPU than for discret GPU.

APIs Used

This sample uses the following OpenCL host functions:

- `clGetDeviceInfo`
- `clCreateBuffer`
- `clCreateImage`
- `clSetKernelArg`
- `clEnqueueNDRangeKernel`
- `clEnqueueMapBuffer`
- `clEnqueueUnmapMemObject`

Run And Controlling the Sample

The sample executable is a console application without any input parameters. The sample initializes OpenCL by looking for GPU device on the Intel platform. If there is no such device, the sample exits with an error message. Otherwise the image processing pipeline executes and the result saves as BMP file that can be opened to see the result.

References

http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf

http://en.wikipedia.org/wiki/Gamma_correction

http://en.wikipedia.org/wiki/Transformation_matrix

https://software.intel.com/en-us/intel-vtune-amplifier-xe