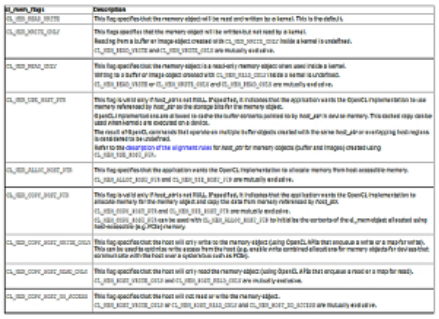
OpenCL Basics: Flags for the creating memory objects



In OpenCL large memory objects, residing in the main memory of the host or the global memory at the accelerator/GPU, need special treatment. First reason is that these memories are relatively slow. Second reason is that the most times serial copy of objects between these two memories take time.

In this post I'd like to discuss all the flags for when creating memory objects, and what they can do to assist in this special treatment.

This is explained on this page of clCreateBuffer in the specifications, but I think it is not really clear. The function clCreateBuffer (and the alike functions for creating images, sub-buffers, etc) suggests that you create a special OpenCL-object to be given as argument to the kernel. What actually happens is that space is made available in main memory of the accelerator and optionally a link with host-memory is made.

The flags are divided over three groups: device access, host access and host pointer flags.
1: Device Access Flags
When defining a kernel, you define each memory-object as read-only, write-only or read-and-write. The memory-objects you create must be of the same. Just like with the kernels, you can pick one:

<empty>. Same as CL_MEM_READ_WRITE.
CL_MEM_READ_ONLY: Kernel can only read from the memory object. Write from the memory object is undefined.
CL_MEM_WRITE_ONLY: Kernel can write to memory object. Read from the memory object is undefined.
CL_MEM_READ_WRITE: Kernel can read and write to the memory object.

The beginner's mistake is to read these flags from the perspective of the host, as they are defined in host-code.

2: Host Access Flags

In case of "CL_MEM_READ_WRITE" it is completely unpredictable what will be done ate the host. Describing how the host will use the memory-object would give compiler-hints for possible optimisations, especially when working with PCIe.

Flags to specify this were introduced in OpenCL 1.2. You can pick one:

<empty>. The host will read and write to this object.
CL_MEM_HOST_WRITE_ONLY. The host will only write to the memory object.
CL_MEM_HOST_READ_ONLY. The host will only read the memory object.
CL_MEM_HOST_NO_ACCESS. The memory-object is used as buffer-object between two or more kernels (should be specified as CL_MEM_READ_WRITE).

Personally I'd suggest you only use it with "CL_MEM_READ_WRITE".

Officially these flags can also be used in combination with READ-ONLY and WRITE-ONLY:

CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_READ_ONLY
CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_WRITE_ONLY

Please let me know, if you have encountered cases where one of these two combinations resulted in faster transfers.

3: Host Pointer Flags

Where it gets interesting (and messy) is when is defined how host and device memory interact. This we can describe with host-pointer flags.

First let have a look on the normal way to send over a buffer of floats to the device:
[raw]cl_mem cl_input = clCreateBuffer(context, CL_MEM_READ_ONLY,
sizeof(float) * count, NULL, NULL);
int err = clEnqueueWriteBuffer(commands, cl_input, CL_TRUE,
0, sizeof(float) * dataCount, p_input, 0, NULL, NULL);
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &cl_input);[/raw]
Here p_input is a pointer to an array of floats, cl_input an OpenCL memory object.

Let us see the definitions of the flags, which influences how data is transferred between host and device.

<empty>. Normal behaviour, with need for explicit writing and reading buffers, as in the above example.
CL_MEM_ALLOC_HOST_PTR. Allocate memory at device, accessible from host. Used for shared memory devices (only ARM MALI has support, afaik).
CL_MEM_COPY_HOST_PTR. Allocate memory at device and initialise with data at host_ptr.
CL_MEM_USE_HOST_PTR. Allocate memory at device and pin it to host_ptr**.**

NB: Even if not explicetly defined, the host access flag CL_MEM_COPY_HOST_NO_ACCESS cannot be used in combination with a host pointer flag.

Let OpenCL handle the copy

The flag CL_MEM_COPY_HOST_PTR results in only one big difference with the default. It does the allocation of device-memory and copying to the device in one step. The code looks like:
[raw]cl_mem cl_input = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(float) * count, p_input, NULL);
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &cl_input);[/raw]
It is a matter of taste, what you use (and I personally don't use this version). It has no advantages for faster transfer, known to me. This flag is not meant for getting data back.

There are cases known that buffer-object 'p_input' in this example cannot be used by clEnqueueWriteBuffer or clEnqueueReadBuffer; in that case **CL_MEM_ALLOC_HOST_PTR** needs also be specified too. Be careful with this, as this is a contradictive contract.

Pinned memory

The flag CL_MEM_USE_HOST_PTR enables the so called "pinned memory". Pinned memory makes it possible to use DMA-transfers over PCIe, which is much faster.

The idea of pinned memory is that there is an exact copy of an object at both the device and the host. Exclusive access rights is given to or the host or the device.

This is the above example with pinned memory:

```
[raw]cl_mem cl_input = clCreateBuffer(context, CL_MEM_READ_ONLY |
   CL_MEM_USE_HOST_PTR, sizeof(float) * count, p_input, NULL);
void* p_map_input = clEnqueueMapBuffer(queue, cl_input, CL_TRUE,
   CL_MAP_READ, 0, sizeof(float) * count, 0, NULL, NULL, &err);
// here we can write data to the buffer from the host
clEnqueueUnmapMemObject(queue, cl_input, p_map_input, 0, NULL, NULL);
// here all changes have been sent back to the device
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &cl_input);
clEnqueueNDRangeKernel(queue, dimension, NULL, global_size,
    local_size, 0, NULL, NULL);
err = clReleaseMemObject(input);[/raw]
```

With clEnqueueMapBuffer the access rights are given to the host. With clEnqueueUnmapMemObject the exclusive access rights to the memory-objects are given to the device.
For reading from the device, most noticeable difference is CL_MAP_WRITE. For read&write, use "CL_MAP_READ | CL_MAP_WRITE". This flag triggers memory-transfers:

<empty>. undefined
CL_MAP_WRITE. Will not do any transfers during mapping, will copy data back to the host during unmapping.
CL_MAP_READ. Will copy data to the device during mapping, will not do any transfers during unmapping.
CL_MAP_READ | CL_MAP_WRITE. Will do transfers during both mapping and unmapping.

Note that the function clEnqueueMapBuffer has many options for blocking and non-blocking execution - so above example might not be optimal for your case.

SoCs: CPUs with embedded GPU

If host and device are the same, what to do then? Best is if the pointer to one memory-area could be shared. Problem is that this changes the whole idea of the current computer-model and this takes time. Say you have a long program and you want the GPU to compute on a part of a buffer - this implies that the GPU should have temporary full access to CPU-memory. This is potentially very dangerous (GPU-viruses) and will not become available over night.

Best is to use pinned memory. If the new models have been worked out, the code can be faster without any code-change. Now it copies memory over at the speed of main memory (25GB/s).

Update: ARM MALI supports exactly this. You need to set the flag **CL_MEM_ALLOC_HOST_PTR**.

More to learn

AMD has introduced a new flag to be added with CL_MEM_USE_HOST_PTR: CL_MEM_USE_PERSISTENT_MEM_AMD. It claims faster transfer-speeds under Windows 7 only. I expect the usage will be merged with CL_MEM_USE_HOST_PTR.

Next step is figuring out asynchronous and non-blocking transfers. Have fun!