

OpenCL™ API

OpenCL™ Tutorial, PPAM 2009

Dominik Behr | September 13th, 2009



Host and Compute Device

The OpenCL™ specification describes the API and the language.

The OpenCL API, is the programming API available on **Host**, which is Windows® or Linux® PC, embedded system, etc. OpenCL API is available as a dynamically linked library or shared object (.DLL, .so).

The OpenCL Language is the programming language based on C99 used to write programs and kernels running on **Compute Device**.



OpenCL™ API

OpenCL API is used by host program to manipulate OpenCL objects, create and destroy contexts, command queues, programs and kernels, device memory (buffers and images).

It is also used to dispatch kernels on compute devices and copy data between the host and the compute device.

The libraries, headers and runtime for OpenCL API are provided in ATI Stream SDK v2.0 beta.



Platforms

There may be more than one OpenCL™ platform available in system installed OpenCL library.

You can query them with `clGetPlatformIDs()`.

And query their properties with `clGetPlatformInfo()`.

This allows OpenCL implementations (platforms) from multiple vendors live under the same roof.

The default platform ID is NULL and if you pass NULL to functions that take `cl_platform_id`, the behavior is implementation defined.



Devices

There can be more than a single device on a platform.

The list of device IDs can be queried with `clGetDeviceIDs()`. You can filter the devices using CPU, GPU and ACCELERATOR flags.

Then query the device properties using `clGetDeviceInfo()`.

For instance:

- **MAX_COMPUTE_UNITS**
- **MAX_WORK_ITEM_SIZES**
- **MAX_WORK_GROUP_SIZE**
- **MAX_MEM_ALLOC_SIZE**
- **GLOBAL_MEM_SIZE**
- **LOCAL_MEM_SIZE**



Contexts

OpenCL™ contexts are created from one or more devices using `clCreateContext()`.

You may mix different device types in a context. Not necessarily a good idea though.

Programs, kernels, buffers, images, events, command queues, are all created in the context.

Buffers, images and events are shared by all devices in the context.



Programs and Kernels

OpenCL™ programs may be created with source using `clCreateProgramWithSource()` or loaded as binaries using `clCreateProgramWithBinary()`. In either case this must be followed by `clBuildProgram()`. Program binaries may be just intermediate representation.

The OpenCL compiler may be a dynamically linked library, in which case calling `clUnloadCompiler()` may help free some memory.

The kernels in program have to be created by name using `clCreateKernel()` function or all of them using `clCreateKernelsInProgram()`.

Building programs and creating kernels may take considerable time.



Memory

There is no malloc. Global memory is allocated using `clCreateBuffer()`.

MEM_READ_ONLY, MEM_WRITE_ONLY flags are important.

host_ptr can be very useful to initialize contents of the buffer. Also **USE_HOST_PTR** or **ALLOC_HOST_PTR** will guarantee that the buffer can be mapped.

Images can be allocated using `clCreateImage{2D|3D}()`.

And all are objects freed using `clReleaseMemObject()`.

Memory objects are created on all devices that belong to a context.



Command Queues

To run commands (kernels, memory operations, etc) on a compute device, one has to create a queue with `clCreateCommandQueue()`.

Queues can be in order (default) or out-of-order, which is optional and support for which can be queried from the device.

Also, you can enable profiling while creating a queue (more on that later).

You can also toggle these with `clSetCommandQueueProperty()`

You can create more than single queue per device.



Kernel Arguments

`clSetKernelArg()` is used to set all types of kernel args.

`cl_mem_object` args are passed as pointer to object, except for NULL pointers.

For `__local` pointer args `arg_value` has to be NULL and the pointer is calculated by runtime and the `arg_size` is added to size of statically declared `__local` variables.



NDRange and Running Kernels

Kernels are enqueued with `clEnqueueNDRangeKernel()` and executed on index space defined by 1,2 or 3D grid.

Every kernel instance (work-item) has unique global ID.

Work items are grouped in work-groups, which have unique group IDs. And work items within work-group have local ID.

$\text{num_groups} * \text{local_size} = \text{global_size}$

$\text{local_id} + \text{group_id} * \text{local_size} = \text{global_id}$

$\text{global_size} \% \text{local_size} = 0$

You can also specify the work-group size in the program.

If you don't specify work-group size, runtime will do it for you.



Copying data

`clEnqueueMapBuffer/clEnqueueUnmapMemObject`

`host_ptr` is very useful when allocating.

`clEnqueueReadBuffer`, `clEnqueueWriteBuffer`,
`clEnqueueCopyBuffer`



Events and Synchronization

clFinish, clFlush

Every* command can wait on list of events.

Every** command can generate an event.

You can wait on events created in another queue that is in the same context.

clWaitForEvents – wait for events on host

clEnqueueMarker* – generate event when all prior commands completed

clEnqueueWaitForEvents** – wait for events before continuing

clEnqueueBarrier*,** – wait for prior commands before continuing

* no wait, ** no event



Memories and multiple devices in a context

OpenCL™ runtime will copy/move the memory between instances on multiple devices.

But it is **up to you** to insert proper waits for events to ensure

If kernel Y on device B uses results from kernel X on device A you have to make sure you wait for kernel X to finish using the event.

```
clEnqueueNDRangeKernel(q_dev_a, kernelx, 0, NULL, &syncevent)
```

```
clEnqueueNDRangeKernel(q_dev_b, kernely, 1, &syncevent, NULL)
```

CL will make the copy before kernel Y is run. This copy may be expensive!



Events and profiling

You can get some information about event with `clGetEventInfo()`.

But more importantly, get profiling information (time) with `clGetEventProfilingInfo()` about when command was

- QUEUED – or put into the queue
- SUBMITTED – submitted from the queue to the compute device
- STARTED – started execution on device
- ENDED – ended execution on device

When optimizing kernels it is all about the time from start to end.



Trademark Attribution

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and/or other jurisdictions. OpenCL is trademark of Apple Inc. used under license to the Khronos Group Inc. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.

©2009 Advanced Micro Devices, Inc. All rights reserved.

