OpenCPI OCL Authoring Model Reference

Authors: Chuck Ketcham, Jim Kulp, and Michael Pepe

OpenCPI

Revision History

Revision	Description of Change	By	Date
2.02	Revised to match the current implementation	Michael Pepe	09/09/2011
2.01	Updated to address inline comments	Michael Pepe	06/20/20011
2.00	Edited based on phone conference with Michael Pepe, Jim Kulp	Chuck Ketcham	01/04/2011
1.00	Initial document creation	Chuck Ketcham	12/14/2010

Table of Contents

References

This document depends on several others. Primarily, it depends on the “OpenCPI Generic Authoring Model Reference Manual”, which describes concepts and definitions common to all OpenCPI authoring models. The OpenCPI OpenCL (OCL) worker authoring model is based on the OpenCL specification (version 1.1 as of this writing), written by the Khronos OpenCL Working Group. The OCL authoring model requires that worker authors be familiar with a subset of the OpenCL specification. Specifically, the OCL worker author is only responsible for writing the worker which in OpenCL parlance equates to a kernel. Therefore, section 6 of the OpenCL specification titled “The OpenCL C Programming Language” is really the only relevant portion of the OpenCL specification for an OCL worker author. That said, general familiarity with the entire OpenCL is certainly helpful.

The OCL worker authoring model is related to the OpenCPI Resource-constrained C Language (RCC) worker authoring model to the extent that both models are designed to be appropriate for resource-constrained processors and both models are also suitable for use on any general-purpose processor with an appropriate compiler (C compiler for RCC and an OpenCL compiler for OCL).

Aside from the RCC model being based on the C language and the OCL model being based the OpenCL language the primary difference between the two models is the OCL model is intrinsically data-parallel as a result of its use of the OpenCL language. While the RCC model does not have an intrinsic data or task parallel structure.

Table 1 - Table of Reference Documents

Title	Published By	Link
OpenCPI Generic Authoring Model 1.01	OpenCPI	http://www.opencpi.org/doc
OpenCL Specification 1.1	Khronos Group	http://www.khronos.org

Introduction

Purpose

The purpose of this document is to define the OpenCPI OCL Authoring Model. The term “OCL” refers to OpenCL (Open Computing Language), which is an open standard for general purpose parallel programming across CPUs (Central Processing Units), GPUs (Graphics Processing Units), and other processors. The software code for an OCL worker is written for execution in an OpenCL kernel, which requires the use of the OpenCL C Programming Language (described in the Khronos specification), which is based on the ISO/IEC 9899:1999 C language specification with specific extensions and restrictions. This model is appropriate for parallel programming tasks involving GPUs or (less optimally) multi-core CPUs that are supported by an OpenCL library and compiler implementation.

The basis of this specification is the OpenCPI Generic Authoring Model Reference (AMR) that defines concepts and metadata common to all OpenCPI authoring models. That document is a prerequisite for this one.

This document introduces and describes the interfaces between the worker and its environment, and the semantics of those interfaces. It also defines the OCL OWD – the XML metadata that describes an OCL worker. The goal of this document is to enable the author of an OCL worker to write the code and the OWD for the OCL worker.

Requirements

- Availability of an OpenCL library implementation. The following are available as of this writing:

Vendor	Hardware supported	Link
NVIDIA	CUDA Architecture GPUs	http://www.nvidia.com/object/cuda_opencl_new.html
AMD	AMD GPU cores and x86 cores	http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx
IBM	IBM Cell and Power architectures	http://www.alphaworks.ibm.com/tech/opencl
Intel	CPUs that support Intel SSE 4.1 or higher	http://software.intel.com/en-us/articles/intel-opencl-sdk/

- Compliance with the OpenCPI Generic Authoring Model Reference Manual (AMR)
- Compliance with the OpenCL C Language (described in the chapter titled "The OpenCL Programming Language" in the Khronos specification).
- Compliance with the OCL local services (described in this document). This may require a port of the OpenCPI core software to the target platform.

Goals

This document defines, for OpenCPI OCL developers, *which* choices can be made by the OCL worker designer about Control, Data, and Local Service interfaces, *how* those choices are described, and how the code must be written, consistent with those choices, to interface the worker to its environment. The intent is to be precise and complete.

Non-Goals

This document is not intended as a usage tutorial for OCL developers. It is a reference specification. It is recognized that separate documents for application developers are needed to reduce the learning and knowledge required to use this model.

Overview

OCL workers are hosted in a *container*, which is responsible for (1) loading, executing, controlling, and configuring the worker, (2) effecting data movement to and from the data ports of the worker, and (3) providing interfaces for the local services available to OCL workers -. The OpenCPI OCL container is supplied by the OpenCPI library and consists of code that runs both on the "host" processor and in the OpenCL kernel environment where the OCL worker code is executed .

When OCL workers are collocated, the container can make use of zero-copy to move data between them. For connections between workers in different containers, the containers move data between each other using some common data transport mechanism. Containers make use of the optimum data transport between the two devices.

The following sections define these aspects of the OCL authoring model:

- the execution model: how the container executes the OCL worker
- the local services: how the worker uses local services and which ones are available
- container-to-worker interfaces: how the container calls the worker's entry points
- worker-to-container interfaces: how the worker calls the container's entry points

OCL Execution Model

The container executes workers such that:

- All OpenCL work-items of a kernel are launched by the container across one OpenCL device. The number of work items, along with their corresponding index space is defined in the OCL OWD XML metadata. See the glossary section of this specification for the definition of kernel and work-item.
- Only one kernel can be executing at a time for a worker. A worker's execution consists of a sequence of OpenCL kernel executions in which the OCL worker code will be called.
- Execution of a worker is enabled when a control operation is requested or when the worker is in the "operating" state and a specified combination of ports is "ready", or a specified amount of time has passed. Control operations are run as a single work-item.
  A "run" method within the worker is called when ports are ready or time has passed.
  Other than explicitly requested control operations, worker execution is a series of "runs" initiated by container logic.
- Workers declare a "run condition" as a simple logical "or" of a small number of masks.
  Each mask indicates a logical "and" of "port readiness bits" (1 << port ordinal).
  An input port is ready when there is an input message for the worker to look at.
  An output port is ready when there is an output buffer that can be filled by the worker.
  The default run condition is one mask of all ones, meaning "run me if all my ports are ready".

If a port being ready is not required for the run condition, the worker can still test its readiness. Otherwise it can assume the readiness of ports indicated in its run condition and not bother checking. A timeout is also provided in the run condition.

- Workers indicate which ports should “advance” at the end of each run, before returning. Advancing a port means releasing (input) or sending (output) the current buffer and implicitly requesting another one. All work-items in a kernel execution must advance the same ports.

The OCL “worker interface” is called by the container, and implemented by the worker. The worker provides its methods (entry point functions) to the container. A “worker context object” is provided to the worker by the container as the first argument to each worker method, analogous to the implied “this” argument to object methods in C++. This context object is a data structure with some advertised structure members (rather than an explicit structure definition: this is similar to the way POSIX defines public structure members in C APIs, also analogous to public member data in C++). This structure also provides the worker with access to port-specific state, any worker-requested memory resources, and the data structure containing its configuration properties.

Execution based on events

The container calls the worker's “run” method when the worker’s “run condition” is true, based on the availability of input buffers (with data) or output buffers (with space) or the passage of time. The worker’s run method executes, processes any available inputs and outputs, indicates when messages are completely consumed as input or produced as output, makes any changes to the “run condition”, and returns. Workers never block. The container conveys the messages in buffers between collocated workers as well as into and out of the container as required by the application assembly’s connections.

Sending or receiving messages via ports

The worker can only indicate that an input buffer has been processed and/or an output buffer has been filled in one execution of the run method. It must return from the run method in order to implicitly wait for more buffers to process. The worker never blocks.

Buffer management

Port Buffer Management

The Worker Interface is designed to have the container provide and manage all buffers. Thus input ports operate by the container providing buffers to the worker filled with incoming messages, and the output buffers operate by the container providing buffers for the worker to fill with messages before being sent. Output buffers are either:

- obtained for a specific output port (since they may be in a special memory or pool specific to an particular output hardware path), *or*
- originally obtained from an input port and passed to output ports, possibly with no copying (the container will copy data as necessary).

Logically, there are three operations performed with buffers, which is the basis for the specific APIs defined later. They are all non-blocking functions:

- ***Request*** that a new buffer be made available. For an input port, it is made available and filled by the container with a new input message. For an output port, it is to be filled by the worker with a new output message. In both cases the ownership of the buffer passes from container to worker when it becomes available. The new buffer may or may not be immediately available after a request.
- Release a buffer to be reused, with its contents discarded. The ownership passes from worker to container. Input buffers must be released (or sent) in the order received on a port, i.e. ownership of input buffers must be passed from worker to container in the order that ownership was given from container to worker, per port.
- Send (enqueue) a buffer on an output port, to be automatically released (later) after the data is sent. The ownership passes from worker to container. If the buffer was originally obtained from an input port, it must be sent or released in the order received from that port.

The concept of "current buffer" of a port exists to support a convenient model for workers that need no special buffer management. A port is *ready* if it has a current buffer. A current buffer on an input port is available to read data from. A current buffer on an output port is available to write data into. The concept of "advancing" a port is simply a combination of releasing (input) or sending (output) the current buffer of the port, and requesting a new buffer to be made available on that port, to become the current buffer when it becomes available in the future. So simple workers wait for ports to be ready (to have buffers), process input buffers into output buffers, advance input and output ports, and return. The interface is designed to make this common model as simple as possible. Workers wait for some ports to be ready, and advance ports after processing buffers.

Several more advanced buffer management requirements are supported by adding extra capabilities for certain situations as needed:

- To support sliding window algorithms, worker are allowed to own previous buffers (not releasing them), while new ones are requested; i.e. allow selective (in order) explicit input buffer release, not implicitly just the most recent buffer obtained. The worker must still release the buffers in the order received.
- To support zero copy from input ports to output ports, allow the worker to send a buffer obtained from an input port to an output port, and not require an empty current buffer to fill on the output port. Such buffers must be sent (or released) in the order received.

The above features are only needed in certain cases, and can be ignored entirely for most simple workers. Supporting them results in support for separate non-blocking interfaces for releasing, sending, and requesting buffers.

Local Memory

OpenCL "local memory" is much faster than "global memory" – it is generally on-chip. It is used to enable coalesced accesses, to share data between work items in a work group, and to reduce accesses to lower bandwidth global memory. To statically allocate local memory space, the `__local` keyword can be used in worker source code for array declarations.

Global Memory

OpenCL global memory generally is the largest capacity memory subsystem on the compute device. It is large and visible to all work groups running on the device. Data placed in global memory is persistent across kernel invocations. I/O buffers and persistent “local memory” buffers are allocated from the global memory pool.

Local Persistent Memory

Unlike the OpenCL local memory that is statically allocated within a kernel using the `__local` keyword the local “persistent” memory comes from the global memory pool. The contents of a local “persistent” memory allocation persists across invocations of a kernel. Local “persistent” memory is requested in the worker’s implementation specific metadata and its delivered to the worker as a named member of the worker’s context structure.

The OCL Worker Interface

The methods in the interface described below are as if it was a C++ or IDL interface, but it is a C interface, with an initial explicit argument to each method that provides a worker context structure, analogous to the implicit "this" argument in C++.

Containers shall use, and OCL Workers shall implement, the following Worker C-language interface, defined in a header file named “`OCL_Worker.h`”, and reserve the prefix “OCL” for compile-time constants and types defined in this file.

The Worker interface consists of control operation methods whose behavior is defined in the AMR, plus an additional ***`run`*** method that supports the event-driven execution model defined above. The ***`run`*** method is the only required method. All the other methods are optional. All processing of the worker occurs in the context of these operations. All operations take a pointer to an ***`OCLWorker<WorkerName>`*** structure. Furthermore, all OpenCL work items for the OCL worker point to the same ***`OCLWorker<WorkerName>`*** object, and therefore this object does not provide a reentrant context to the worker’s OpenCL work items executing in parallel.

All defined data types use the common prefix “OCL”.

The interface uses several basic integer types consistent with their CORBA C/C++ language mapping, to provide some compiler independence. The integer types are defined using the ISO C 99 `<stdint.h>` types. These basic types are: `uint16_t` and `uint32_t`. The type `OCLBoolean` is aliased to `uint32_t`, to meet the minimum size requirement for OpenCL kernel arguments. The `OCLOrdinal` type is an alias for `uint32_t`, and is used when ordinals are required (ports, operations, exceptions, properties).

OCL Interface Data Types

OCLResult

The `OCLResult` type is an enumeration type used as the return value for all worker methods. It indicates to the container what to do when the worker method returns, as follows:

- `OCL_OK`: worker operation succeeded without error
- `OCL_ERROR`: operation did not succeed, but error is not fatal to worker or container, thus the operation may be retried.
- `OCL_FATAL`: worker should never run again; it is non-functional; the container or other workers may be damaged; the worker is in an **Unusable** control state as defined in the AMR. Note that the container may know that it, or other workers are protected from damage, but the portable worker indicates this condition in case there is no such protection.
- `OCL_DONE`: worker needs no more execution; a normal completion. See usage below.
- `OCL_ADVANCE`: worker is requesting that all ports be advanced (run method only).

These result values apply to each method as defined in their specific behavior. Some values are not valid results for all methods. When the result is `OCL_ERROR` or `OCL_FATAL`, the worker may set the *`errorString`* member of the `OCLWorker` structure to non-NULL to add a string description to the error. See the `OCLWorker` type below.

OCLPortMask

The `OCLPortMask` type is a bit mask used to indicate a boolean value for each port. It is an integer type where a given bit being == 1 (1L << port_ordinal) means the value for that port is TRUE.

OCLRunCondition

The `OCLRunCondition` structure type holds the information used by the container to determine when it is appropriate to invoke the *run* operation of a worker. The defined members are always written by the worker, and never by the container.

`OCLRunCondition` Member Name	**Member Data Type**	**Member Description**
`portMasks`	`OCLPortMask`	A a bit-mask of port readiness. The run condition is considered "true" when the masks is "true". The mask is "true" when all indicated ports are ready (logical AND of port readiness). A port is indicated by its bit being set (1 << port_ordinal). The default port mask requires all ports to be ready.

timeout	OCLBoolean	Indicates that the *usecs* member is used to determine when enough time has passed to make the run condition true. Thus this value can used to enable or disable the timeout, without changing *usecs*.
usecs	uint32_t	If this amount of time has passed (in microseconds) since the run operation was last entered, the run condition is true.

Thus the overall run condition is the logical OR of the portMasks and the timeout.

```
typedef struct
{
  OCLPortMask portMasks;
  OCLBoolean timeout;
  uint32_t usecs;

} OCLRunCondition;
```

OCLMethod

The `OCLMethod` type is simply a convenience type declaration to declare worker methods (in C) that do not have any arguments other than the `OCLWorker` structure. It declares a function type that takes a worker's context structure as the single input argument, and returns an `OCLResult` value. The `<WorkerName>` is the name of the worker specified in the worker's metadata. The worker's name is used as a prefix on the the worker's method names. It is defined as follows:

```
typedef OCLResult <WorkerName>OCLMethod ( OCLWorker<WorkerName>* self );
```

OCLRunMethod

The *`OCLRunMethod`* type specifies the signature of the worker's *run* method. The method is described below. The type is defined as follows:

```
typedef OCLResult <WorkerName>OCLRunMethod ( OCLWorker<WorkerName>* self,
                                              OCLBoolean timedout,
                                              OCLBoolean* newRunCondition );
```

OCLBuffer

The `OCLBuffer` type is a structure that holds information about a buffer. It has the members described in the table below, all of which are written by container functions and not by the worker. There may be other undocumented members.

`OCLBuffer` **Member Name**	**Member Data Type**	**Member Description**
`data`	`__global void* const`	A `const` pointer to the data in the buffer. When this member of the structure is NULL, there is no buffer.
`maxLength`	`const uint32_t`	The maximum number of bytes that may be placed in the buffer (maximum message length). This is *not* the actual length of the valid data in the buffer.

OCLPortAttr

The *`OCLPortAttr`* type is a structure that contains the attributes of a worker's port. There may be other undocumented members.

`OCLPortAttr` **Member Name**	**Member Data Type**	**Written by**	**Member Description**
`optional`	`OCLBoolean`	container	Indicates is this is an optionally connected port.
`connected`	`OCLBoolean`	container	Indicates if this port is currently connected or not.

`length`	`uint32_t`	container/worker	For input ports, the number of valid bytes in the message the container has placed in the current input buffer. For output ports, the number of valid bytes in the message the worker has placed in the current output buffer.
`u.exception`	`const OCLException`	container/worker	The exception ordinal of the message in the current buffer. Zero indicates no exception.
`u.operation`	`OCLOperation`	worker	The operation ordinal of the message in the current output buffer.

OCLPort

The *`OCLPort`* type is a structure that contains the current state of a worker's port. The container is free to define this structure with any content and any ordering as long as the documented members are supported. (This style of structure standardization is from POSIX). The members are written by either the container or the worker, but never both. Members written by the container are declared "`const`" to enhance error checking when compiling worker implementations. For code readability and reliability, there is a union in the structure that aliases two substructures with members used differently for input ports and output ports. Members used for input ports only are prefixed with "input.". Members used for output ports only are prefixed with "output.".

`OCLPort` **Member Name**	**Member Data Type**	**Written by**	**Member Description**

`current`	`OCLBuffer`	container	The current buffer to use for the port. The actual format of the message in the buffer depends on which request, response, or exception message is represented. The value of the *data* member is NULL if there is no current buffer. Port readiness is equivalent to the current.data != NULL.
`attr`	`OCLPortAttr`	container/ worker	Attributes of the port.

OCL Container Functions

Since OpenCL does not support function pointers the OCL container function are a set of external symbol definitions: rather than a neat collection of function pointers stored inside of worker context.

The use of these container methods is optional, and entirely unneeded for simple workers that only use run conditions and the *run* method. They are all used to provide additional flexibility and functionality in buffer handling.

SPOCL Container Function Prototype	**Container Function Description**
`void OCLRelease ( OCLBuffer* buffer );`	Release a buffer for reuse. If the buffer is a current buffer for a port, it will no longer be the current buffer. Buffer ownership passes back to the container. Non-blocking. Must be done in the order obtained, per port.
`void OCLSend ( OCLPort* port, OCLBuffer* buffer, OCLOrdinal op, uint32_t length );`	Send a buffer on an *output* port. If the buffer is a current buffer for a port, this buffer will no longer be the port's current buffer. The "op" argument is an operation or exception ordinal. Buffer ownership passes back to the container. Non-blocking.
`OCLBoolean OCLRequest ( OCLPort* port, uint32_t max );`	Request a new buffer be made available as the current buffer on a port. An optional (non-zero) length may be supplied. If the port already has a current buffer, the request is considered satisfied. The return value indicates whether a new current buffer is available. Non blocking.

`OCLBoolean OCLAdvance ( OCLPort* port, uint32_t max );`	Release the current buffer *and* request that a new buffer be made available as the current buffer on a port. An optional (non-zero) length may be supplied. This is a convenience/efficiency combination of release-current-buffer request. The return value indicates whether a new current buffer is available. Non blocking.
`void OCLTake ( OCLPort* port, OCLBuffer* releaseBuffer, OCLBuffer* takenBuffer );`	Take the current buffer from a port, placing it at *takenBuffer. If `releaseBuffer != NULL`, first release that buffer. Non-blocking. Ownership is retained by the worker. The current buffer taken is no longer the current buffer. Used when the worker needs access to more than one buffer at a time from an input port.

OCLWorker<WorkerName>

This structure type (typedef) represents the visible state of a worker. It is passed by reference to every method, analogous to the implicit “this” argument in C++. The container is free to define this structure (in `<WorkerName>_Worker.h`) with any content and any member ordering as long as the documented members are supported. The members are written by either the container or the worker, but not both. Members written by the container are declared "const" to enhance error checking when compiling worker implementations.

`OCLWorker` **Member Name**	**Member Data Type**	**Written by**	**Member Description**
`properties`	`__global <WorkerName>Properties* const`	container	A const pointer to the properties structure for the worker, whose layout is implied by the properties of the implementation. The value may be NULL if there are no such properties.
`<LocalMemoryNames>`	`__global void* const`	container	Zero or more persistent local memory buffers. The names of these buffers come from the worker’s implementation metadata.

`runCondition`	`OCLRunCondition*`	worker	Initialized from the `OCLDescriptor` `runCondition` member. Checked by container after calling the start method.
`<PortNames>`	`OCLPort`	varies by member	One or more `OCLPort` structures defined above, and named using the names provided in the worker's OCS.

Methods called by container, implemented by worker

initialize

This method implements the OpenCPI *initialize* control operation. See AMR for rationale and behavior. It is optional. No implementation is required.

Synopsis

```
OCLResult <WorkerName>_initialize ( __global OCLWorker<WorkerName>*
self );
```

Returns

This method shall return an `OCLResult` value.

Exceptions/Errors

If the initialization cannot succeed, it shall return `OCL_ERROR`. If the worker detects an error that would disable the implementation or its environment, it shall return `OCL_FATAL`. Otherwise it shall return OCL_OK if normal worker execution should proceed.

start

This method implements the OpenCPI *start* control operation. See AMR for rationale and behavior. It is optional. No implementation is required.

Synopsis

```
OCLResult <WorkerName>_start ( __global OCLWorker<WorkerName>* self );
```

Returns

This method shall return an `OCLResult` value.

Exceptions/Errors

If the *start* method cannot succeed, it shall return `OCL_ERROR`. If the worker detects an error that would disable the implementation or its environment, it shall return `OCL_FATAL`. Otherwise it shall return `OCL_OK` if normal worker execution should proceed. It shall return `OCL_DONE`, if no worker execution should proceed. Returning `OCL_DONE` indicates to the container that the worker will never require execution, and provides this advice to the container to allow the container to take advantage of this fact and possibly release resources (such as I/O buffers) prior to the worker being destroyed or reused.

stop

This method implements the OpenCPI *stop* control operation. See AMR for rationale and behavior. It is optional. No implementation is required.

Synopsis

```
    OCLResult <WorkerName>_stop ( __global OCLWorker<WorkerName>* self );
```

Returns

This method shall return an `OCLResult` value.

Exceptions/Errors

If the *stop* method cannot succeed, it shall return `OCL_ERROR`. If the worker detects an error that would disable the implementation or its environment, it shall return `OCL_FATAL`. Otherwise it shall return `OCL_OK`.

release

This method implements the OpenCPI *release* control operation. See AMR for rationale and behavior. It is optional. No implementation is required.

Synopsis

```
    OCLResult <WorkerName>_release ( __global OCLWorker<WorkerName>* self
    );
```

Returns

This method shall return an `OCLResult` value.

Exceptions/Errors

If the *release* method cannot succeed, it shall return `OCL_ERROR`. If the worker detects an error that would disable the implementation or its environment, or it is unable to return resources it allocated, it shall return `OCL_FATAL`. Otherwise it shall return `OCL_OK`.

afterConfigure

This method implements the OpenCPI *afterConfigure* control operation. See AMR for rationale and behavior. It is optional. No implementation is required.

Synopsis

```
	OCLResult <WorkerName>_afterConfigure ( __global OCLWorker<WorkerName>*
	self );
```

Returns

This method shall return an `OCLResult` value.

Exceptions/Errors

If the *afterConfigure* method cannot succeed, it shall return `OCL_ERROR`. If the worker detects an error that would disable the implementation or its environment, it shall return `OCL_FATAL`. Otherwise it shall return `OCL_OK`

beforeQuery

This method implements the OpenCPI *beforeQuery* control operation. See AMR for rationale and behavior. It is optional. No implementation is required.

Synopsis

```
	OCLResult <WorkerName>_beforeQuery ( __global OCLWorker<WorkerName>*
	self );
```

Returns

This method shall return an `OCLResult` value.

Exceptions/Errors

If the *beforeQuery* method cannot succeed, it shall return `OCL_ERROR`. If the worker detects an error that would disable the implementation or its environment, it shall return `OCL_FATAL`. Otherwise it shall return `OCL_OK`.

run

Brief Rationale

The *run* method requests that the worker perform its normal computation. The container only calls this method when the worker's *run condition* is true. This allows the container to fully manage all synchronization, minimizing the resources in use when the worker is not able to run.

Synopsis

```
OCLResult <WorkerName>_run ( __global OCLWorker<WorkerName>* self,
                              OCLBoolean timedout,
                             __global OCLBoolean* newRunCondition );
```

Behavior

The *run* method shall perform the worker's computational function and return a result. The *run* method may use information in its property structure, state of its ports, and its private memory to decide what to do. Normally this involves using messages in buffers at input ports to produce messages in buffers at output ports.

The `timedout` input parameter indicates whether the *run* method is being invoked due to time passing (the *usecs* value of the run condition). This indication is independent of port readiness.

Each port's `current.data` member may be tested to indicate port readiness, although no testing is required for ports in the run condition, when `timedout` is false (or it is not in the run condition at all).

The *run* method can indicate that *all* ports should be advanced by returning a special value. It can also indicate disposition of buffers and ports by using the ***release***, ***send***, ***request***, or ***advance*** container functions. Each function may only be called once per port per execution of the run method.

The *run* method may change the run condition by writing a TRUE value to the location indicated by the `newRunCondition` output argument, and setting a new run rondition in the `newRunCondition` member of `OCLWorker`. The worker maintains storage ownership of all run conditions.

Returns

This method shall return an `OCLResult` value.

The value of `OCL_ADVANCE` shall indicate that all ports should be advanced that were ready on entry to the *run* function and were not subject to container functions called since then (e.g. advance, request etc.). This is a very common case.

Exceptions/Errors

If the *run* method cannot succeed, it shall return `OCL_ERROR`, indicating that it should not be called again (run condition should be considered always false). If the worker detects an error that would disable the implementation or its environment, it shall return `OCL_FATAL`. Otherwise it shall return `OCL_OK` or

`OCL_ADVANCE` if normal worker execution should proceed (and the *run* method called again when the run condition is true). It shall return `OCL_DONE`, if no worker execution should proceed. Returning `OCL_DONE` indicates to the container that the worker will never require execution, but there is no error, and provides this advice to the container to allow the container to take advantage of this fact and possibly release resources (such as I/O buffers) prior to the worker instance being destroyed or reused.

test

This method implements the OpenCPI *test* control operation. See AMR for rationale and behavior. It is optional. No implementation is required.

Synopsis

```
OCLResult <WorkerName>_test ( __global OCLWorker<WorkerName>* self );
```

Returns

This method shall return an `OCLResult` value.

Exceptions/Errors

If the *test* method cannot succeed, it shall return `OCL_ERROR`. If the worker detects an error that would disable the implementation or its environment, it shall return `OCL_FATAL`. Otherwise it shall return `OCL_OK`. Note that test “results” are provided in readable properties, so returning `OCL_ERROR` implies that the test could not be run at all, usually due to invalid test properties. It does not indicate that the result of running the test was not success.

Code generation from OCL OCS and OWD Metadata

Ports inferred from the OCL OCS DataInterfaceSpec element.

The port interfaces for a worker shall be consistent with the associated `DataInterfaceSpec` XML element in the OCL OCS for the worker. OCL worker ports are inferred from the OCL OCS information as follows:

- Ports are ordered according to their appearance in the OCS.
- Ports with protocols with no two-way operations imply one OCL worker port.
- Ports with some two way operations imply two OCL worker ports; first read-only input then write-only output.
- Consumer/Server/Provider ports take as input request messages and (for two-way operations) write response messages as output.

- Producer/Client/User ports write request messages as output and (for two-way operations) read response messages as input.
- All OCL worker port interfaces are read or write, but not both.

Property structure from OCS and OWD Property Elements

The layout of the property structure is derived from the Property elements found in (first) the OCS and (second) the `OCLImplementation` in the OWD. This layout is expressed as a normal C structure, with member names being the properties' "Name" attributes from the OCS and OWD. This `struct typedef` can be used for accessing property values using the "properties" pointer in the *`OCLWorker<WorkerName>`* structure. This struct is automatically generated by OpenCPI tools as part of the automatic generation of the complete `Xyz_Worker.h` file. The rules for generating this structure are:

- The standard name of the structure type (`typedef` name) is `XyzProperties` where "`Xyz`" is the mixed case (usually with initial upper case) worker implementation name in the OCS.
- The standard name for the generated header file containing this definition (and others) is "`Xyz_Worker.h`". This precludes using "OCL" for the worker implementation name.
- Properties which are sequences are preceded by an unsigned long member whose name is the property name with "`_length`" appended. Padding may be added before and after the length member to achieve the required alignment of this length field as well as the sequence data following it.
- Sequence properties are represented by property structure members that are C arrays whose length is the `SequenceLength` attribute from the `Property` element.
- Structure properties have structure tags with the same name as the property name, preceded by the uppercase worker implementation name.
- The names of the integer types for simple properties: `short`, `ushort`, `long`, `ulong`, `octet`, `char` are mapped to the respective `<stdint.h>` types: `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, and `uint8_t`.
- The names of the non-integer types for simple properties are the OCS-defined type names capitalized and prefixed with "`OCL`": e.g. `OCLBoolean`, `OCLChar`, `OCLFloat`, and `OCLDouble`.
- Properties that are string properties are "`OCLChar`" arrays whose size is one more than the `StringLength` attribute of that string property in the `Property` element, and the values are null terminated strings.

Operation and exception ordinals

The `Xyz_Worker.h` file will supply an enumeration for each port, defining opcode or exception ordinals. The ordinals will have the `typedef` name `XyzPqrOperation` or `XyzPqrException`, and the constant name `XYZ_PQR_ORD`, where `ORD` is the name of the operation or exception. For any reply port, `OCL_NO_EXCEPTION` and `OCL_SYSTEM_EXCEPTION` are also valid. `PQR` is the upper-cased port name.

These ordinals represent the message types based on the Operation sub-elements of the `Protocol` sub-element of the `DataInterfaceSpec` corresponding to the port.

Message structures

Since the Argument elements in an Operation element are allowed to be variable length sequences and strings, arguments in operation messages can be variable in length. Thus in the general case, when multiple arguments are unbounded, no C data structure can capture the entire message format. This specification defines a normative structure for message contents up to and including the first variable length argument, and, for structures or unions, the first variable length member. This applies to request, reply and exception messages.

This partial solution to mapping the message contents into a normative structure is intended to facilitate many, but not all message content access. Since fixed size messages, or messages whose only variable element is the last argument, are very common, this solution adds portability value in many cases.

The message structures generated for each port will have the `typedef` name `XyzPqrOpr`. `Xyz` is worker name, `Pqr` is port name, and `Opr` is operation or exception name. The members of the structure are named the same as the property structure above, but with argument names from the `Argument` elements rather than from the `Property` elements. If the first variable length argument is a sequence of variable length types (sequence or string), the type of the member for that variable length type will be `uint32_t`, which will be the last member in the structure.

The message structure member types for the integer types: `short`, `unsigned short`, `long`, `unsigned long`, `long long`, `unsigned long long, octet, and enum` are mapped to the respective CDR-sized `<stdint.h>` types: `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`, `uint8_t` and `uint8_t.`

The member types for the non-integer basic types are the Type attribute value names capitalized and prefixed with "`OCL`": e.g. `OCLBoolean`, `OCLChar`, `OCLFloat`, and `OCLDouble`. String types are "`OCLChar`" arrays.

OpenCPI Implementation Descriptions for OCL Workers (OCL OWD)

Introduction

This section describes the format and structure of the OCL OWD: the XML document that refers to or includes an OpenCPI Component Specification (OCS), and which specifies the implementation-specific aspects of an OCL worker. The AMR defines information common to all OWDs. This section defines aspects of the OCL OWD – specific to OCL workers. The AMR and its descriptions of OCSs and OWDs is a prerequisite for this section.

Top Level Element: OCLImplementation

An `OCLImplementation` element contains information provided by someone creating an OCL worker based on a component specification (OCS). It includes or references an OCS, and then describes implementation information about a particular implementation of that OCS. The `OCLImplementation` must either include as a child element a complete `ComponentSpec`, or include one by reference, for example, if the "`vsadd`" implementation of the "`vsadd_spec`" specification referenced the component specification found in the "`vsadd.xml`" file:

```
<OclImplementation>
  <xi:include href="vsadd_spec.xml"/>
  <ControlInterface Name="vsadd"/>
</OclImplementation>
```

The `OCLImplementation` follows the specification of OWDs in general as specified in the AMR. This section defines the aspects of the OCL OWD (`OCLImplementation`) that is not common to all OWDs.

Attributes of an OCL Component Implementation

Name attribute

The "`Name`" attribute of the component implementation is used to generate the programming language name of the worker (e.g. used in the name of, and the contents of, the OCL generated header file). Per the AMR, when not specified the implementation name defaults to the name of the OWD file itself without directories or extensions.

Control Plane Aspects of an OCL Component Implementation

ControlInterface element

The control interface child element (`ControlInterface`) of the `OCLImplementation` specifies implementation aspects of the worker's control functionality that is different from the default.

As stated in the AMR, `Properties`, `Property`, or `SpecProperty` elements can occur under this element. A `Properties` element can be referenced in an external file via `xi:include`.

ControlOperations attribute of ControlInterface

This attribute is as defined in the AMR. If not specified, no control operation methods are implemented by the OCL worker (only the "`run`" method). This information is used in the generation of the header file. This allows the worker author to have no implementation code at all for the control operations with default behavior (see the AMR).

Data Plane Aspects of an OCLImplementation

Port element

The Port child element of the `OCLImplementation` specifies OCL-specific information about a data interface specified in the OCS. It references a `DataInterfaceSpec` by its `Name` attribute. Thus the `Name` attribute of the `Port` element must match the `Name` attribute of a `DataInterfaceSpec` element of the `ComponentSpec`. The `Port` element adds implementation-specific information about the interface initially defined in that `DataInterfaceSpec`.

Name attribute

This attribute specifies the name used to reference the `DataInterfaceSpec` in the `ComponentSpec` (OCS).

MinBuffers attribute

This numeric attribute specifies the minimum number of message buffers required by the OCL worker for a port. The OCL Worker Interface allows the worker code (typically in the “run” method) to “take” a buffer from a port, and ask for a new buffer for that port while retaining ownership of the previous buffer from that port. This behavior requires that the infrastructure provide at least 2 buffers for that port. Thus this attribute informs the infrastructure as to the minimum buffering requirements of the OCL worker implementation for that port. The default is 1. This attribute should *not* be used to “tune” the buffer count for performance but only specify the actual minimum requirements for the correct functioning of the OCL worker.

DataValueWidth attribute

This numeric attribute specifies the size in bits of the elements in a data message. This information is used to convert the size in bytes of a buffer to work-items.

OCL Compiler

The OCL compiler compiles worker source code into binary format. The target device must be installed on the build system along with the relevant OpenCL distribution. The compiler creates two files: the binary format file (with a .so extension) and the compiler artifact file (appended with “`_artifact.xml`”).

The command line tool is run as follows:

```
ocpiocl <source file list>  -o <output file name> -I<Include path> -
D<defines>
```

The individual fields are described below:

`<source file list>`: A list of source and header files enclosed in double quotes. Each file within the string is separated by whitespace. Path information may be prepended to each file name. This is a required field.

`-o <output file name>`: The is the name of the output file that the compiler will generate. This is a required field.

`-I <include path>`: Zero or more locations to search for header files. This is an optional field.

`-D <define>`: Zero or more compile time defines. This is an optional field.

Summary of OpenCPI OCL authoring model

- OCL workers are written to implement the Worker interface, called by the container, and optionally use the Container interface, called by the worker. The interface is conveyed to the container via a set of externally defined container functions prefixed with OCL (see the section named OCL Container Functions)
- OCL workers may call the built-in functions defined in section 6.11 of the OpenCL C Language.
- OCL workers use data structures and ordinals as defined in the metadata code generation section (which will be automatically generated from the OWD).
- OCL worker authors must make the following implementation-specific metadata available to the build process (see OWD below) for compiling/linking/loading worker implementation binaries on a given platform:
- Implementation name
  - Which control operations are implemented
  - Which properties require `beforeQuery` or `afterConfigure` notification
  - Minimum number of buffers required at each port (default is one)
  - Data value widith in bytes
  - Local memory name and size in bytes
- The information in the OWD is used (which includes the implementation metadata above), to

drive the build process and the code generation process for the OCL worker's header file.

Code examples

Worker code example

Here is a simple example of an "`vsadd`" worker using the base profile whose:

- initial run condition is the default (run when all ports are ready, no timeout),
- `initialize`, `start`, `stop`, `release` and `test` methods are empty
- one input port (0) with interface `in` (only oneways) and one output port (1) `out` (only oneways)
- one oneway IDL interface operation `Op1` on input (i.e. can ignore "operation"), which is an array of 4096 "floats".
- one oneway IDL interface operation `Op2` on output
- one simple property, called `scalar`, of type `float`.

The worker would have automatically generated types and structures like this (based on OCS and OWD), and put in a file called "`vsadd_Worker.h`":

```
/* This file contains the implementation declarations for worker vsadd */

#ifndef OCL_WORKER_VSADD_H__
#define OCL_WORKER_VSADD_H__

#include <OCL_Worker.h>

#if defined (__cplusplus)
extern "C" {
#endif

/*
 * Property structure for worker vsadd
 */
typedef struct {
  OCLFloat      scalar; // offset 0, 0x0

} VsaddProperties;

/*
 * Worker context structure for worker vsadd
 */
typedef struct {
  __global VsaddProperties* properties;
  OCLRunCondition runCondition;
  __global void* mem0;
  __global void* mem1;
  OCLPort in;
  OCLPort out;

} OCLWorkerVsadd;


#if defined (__cplusplus)
}
#endif
#endif /* #ifndef OCL_WORKER_VSADD_H__ */
```

The actual code for the worker would look like this:

```
#include "vsadd_Worker.h"

/*
 * Required work group size for worker vsadd run() function.
 */
#define OCL_WG_X 64
#define OCL_WG_Y 1
#define OCL_WG_Z 1

/*
 * Methods to implement for worker vsadd, based on metadata.
 */

OCLResult vsadd_run ( __local OCLWorkerVsadd* self,
                      OCLBoolean timedOut,
                     __global OCLBoolean* newRunCondition )
{
   (void)timedOut;
   (void)newRunCondition;

```

```
  const size_t n_elems = self->in.attr.length / sizeof ( float );

  __global const float* src = ( __global float* ) self->in.current.data;

  __global float* dst = ( __global float* ) self->out.current.data;

  int gid = get_global_id ( 0 );

  if ( gid >= n_elems )
  {
    return OCL_DONE;
  }

  dst [ gid ] = src [ gid ] + self->properties->scalar;

  self->out.attr.length = self->in.attr.length;
  self->out.attr.u.operation = self->in.attr.u.operation;

  return OCL_ADVANCE;
}
```

A simple container implementation would simply have a loop, testing run conditions, and calling run methods.

So, on each execution, the worker sees the status of all I/O ports, and can read from current input buffers, and write to current output buffers. It must return to get new buffers, after specifying whether buffers are consumed or filled during the execution.

Note the use of the `OCL_WG_[X|Y|Z]` defines. These defines are required. They are used to specify the local work-group size required by the worker's run() method implementation.

Implementation Details

These details are subject to change and will be moved from this document.

OpenCPI programming model contrasted with OpenCL programming model

OpenCPI

Familiarity with OpenCL is a prerequisite for this section. According to the OpenCL Specification, the OpenCL Platform Model consists of a host connected to one or more OpenCL devices. Using this model, the OpenCL application submits commands from the host to execute computations on the processing elements within a device. This model requires the programmer to write host-side code in which the host provides a context for the execution of kernels (note that a "kernel" is a function declared in a program and executed on an OpenCL device). The host-side code, using this model, must make use of API calls that manage OpenCL objects and allows the execution of commands to launch a kernel or read/write to a memory object. These functions are described in the "OpenCL Runtime" section of the OpenCL specification, but are not necessary to learn or use to write OpenCPI OCL workers.

In contrast to the OpenCL approach, the OpenCPI OCL authoring model does not require the programmer to write host-side code for an OCL worker implementation. The programmer only writes kernel-side code according to this specification (that is not the top level kernel code, but called by hidden top level kernel code supplied by OpenCPI). The OpenCPI OCL worker code uses the "OpenCL C Programming Language" chapter of the OpenCL Specification.

Return from OCL worker method functions

When a worker method function returns, control is passed back to kernel-side container logic. This logic handles the decision of whether or not to terminate the kernel or re-execute a "run" method. It also handles the transfer of internal status information to a global buffer, which would ultimately be transferred to container logic residing on the host. All work-items must return the same value.

Property Access

Introduction

Properties can be read and modified by control application code running on the host and by kernel code running on an OpenCL device. Since the kernel and application programs access separate memory spaces, contents of memory must be mapped or copied as needed by the container logic to maintain data coherence. These map and copy operations are performed according to the following specification:

Property is written by a worker and read by the host

The host-side container logic launches a host-side thread to launch a kernel and then blocks on its termination. When the kernel has terminated, the host-side thread unblocks and executes a map operation to map the kernel property address space into the host address space.

When the host-side control application attempts to read a property value, it calls the "read" container function which returns the value of the property from the host address space. In cases where the kernel has not yet terminated, this value can be stale and not match the value in the kernel address space.

Property is written by a host and read by a worker

When the host-side control application updates a property value, it calls the "write" container function which simply writes the value into the host address space. This value is only copied into the kernel address space when the host-side container launches a kernel. The host-side container logic only launches a kernel when it determines that a worker control method should be executed.

Device Index specification

[This section belongs in the user guide section for this container type].

The OpenCPI Control API function used to find a container has a “model” argument.For OpenCL containers, the “model” argument is “ocl”. At this time the OCL container enumerates all OpenCL devices on a platform. At this time the OCL container find function does not allow a control applicaiton to select a particular OpenCL device.

Glossary

Configuration Properties – Named values associated with a worker that may be read or written by the control application and/or the worker. Their values indicate or control aspects of the worker's operation. Reading and writing these property values may or may not have side effects on the operation of the worker. Configuration properties with side effects can be used for custom worker control. Each worker may have its own, possibly unique, set of configuration properties. Some properties come from the OCS: they are common to all implementations that implement that OCS. Other properties can be added in the OWD that are specific to that particular worker implementation.

Container – An OpenCPI infrastructure element that "contains", manages and executes a set of application workers. Logially, the container "surrounds" the workers, mediating all interactions between the worker and the rest of the system.

Worker – A concrete implementation (and possibly runtime instance) of a component, generally existing within a container. A worker is implemented consistent with its OWD.

Kernel – A function declared in a program and executed on an OpenCL device.

Work-Item – One of a collection of parallel executions of a kernel invoked on a device. A work-item is executed by one or more processing elements as part of a work-group executing on a compute unit. A work-item is distinguished from other executions within the collection by its global ID and local ID.