OpenCL* Device Fission for CPU Performance

Summary

Device fission is an addition to the OpenCL* specification that gives more power and control to OpenCL programmers over managing which computational units execute OpenCL commands. Fundamentally, device fission allows the sub-dividing of a device into one or more sub-devices, which, when used carefully, can provide a performance advantage, especially when executing on CPUs.

The newly released Intel® SDK for OpenCL* Applications 2012 is a comprehensive software development environment for OpenCL applications on 3rd generation Intel® Core™ processor family-based platforms. This SDK also provides developers with the ability to develop and target OpenCL applications on Intel CPUs of previous generations using both the Windows* and Linux* operating systems.

The Intel SDK for OpenCL Applications 2012 provides a rich mix of OpenCL extensions and optional features that are designed for developers who want to utilize all resources available on Intel CPUs. This article focuses on device fission, available as an OpenCL 1.1 extension with this version of the SDK.

Download your FREE copy of the Intel SDK for OpenCL Applications 2012 at:
www.intel.com/software/opencl

What is Device Fission?

The OpenCL specification is composed of a hierarchy of several models including the Platform, Execution, Memory, and Programming Models. The highest level model, the Platform Model, consists of a host processor connected to one or more OpenCL devices. OpenCL devices execute commands submitted to them by the host processor. A device can be a CPU, GPU, or other accelerator device. A device further comprises one or more computational (or compute) units. For example, for a multicore CPU, a computational unit is a thread executing on a core. For a GPU, a computational unit is a thread executing on a stream processor or streaming multiprocessor (SM). As the number of computational units and threads have grown over time, it is beneficial at some point to exert more control over these resources, rather than treating them as a single homogenous computing resource.

To help control which computational units execute OpenCL commands, an important addition, named Device Fission, was made to the OpenCL specification to give more power to the OpenCL programmer. Device fission is defined in the OpenCL 1.2 specification (and is available as an OpenCL 1.1 extension).

Device fission is a useful feature that allows the sub-dividing of a device into two or more sub-devices. Google dictionary defines fission as “the action of dividing or splitting something into two or more parts.” After identifying and selecting a device from an OpenCL platform, you can further split the device into one or more sub-devices.

There are several methods available for determining how sub-devices are created. Each sub-device can have its own context and work queue and its own program if needed. This enables more advanced task parallelism across the work queues.

A sub-device acts just like a device would act in the OpenCL API. An API call with a device as a parameter can have a sub-device as a parameter. In other words, there are no special APIs for sub-devices, other than for creating one. Just like a device, a context or a command queue can be created for a sub-device. Using a sub-device allows you to refer to specific computational units within the original device.

Sub-devices can also be further sub-divided into more sub-devices. Each sub-device has a parent device from which it was derived. Creating sub-devices does not destroy the original parent device. The parent device and all descendent sub-devices can be used together if needed.

Device fission can be considered an advanced feature that can improve the performance of OpenCL code and/or manage compute resources efficiently. Using device fission does require some knowledge of the underlying target hardware. Device fission should be used carefully and may impact code portability and performance if not used properly.

Why Use Device Fission?

In general, device fission allows the programmer to have greater control over the hardware platform by selecting which computational units are used by the OpenCL runtime to execute commands. The reason this control is useful is that, if used properly, it can provide better OpenCL performance or make the overall platform more efficient.

Here are some example cases where device fission is useful.

- Device fission allows the use of a portion of a device. This is useful when there is other non-OpenCL work on the device that needs resources. It can guarantee the entire device is not taken by the OpenCL runtime.
- Device fission can allow specialized sharing among work-items such as sharing an L3 cache or sharing a NUMA node.
- Device fission can allow a set of sub-devices to be created, each with its own command queue. This lets the host processor control these queues and dispatch work to the sub-devices as needed.
- Device fission allows specific sub-devices to be used to take advantage of data locality.

Later in this paper, strategies for using device fission are discussed in more detail, but first we'll show how to code for device fission in OpenCL 1.2.

How to Use Device Fission in OpenCL* 1.2

This section provides an overview on how to use device fission and create sub-devices in OpenCL 1.2. Refer to section 4.3 (Partitioning a Device) of the OpenCL 1.2 specification for further details.

There are several partitioning types and options available when creating sub-devices. The three basic options for determining how to split or partition the device are:

- Equally – Partition the device into as many sub-devices as can be created, each containing a given number of computational units.
- By Counts – Partition the device based on a given number of computational units in each sub-device. A list of the desired number of compute units per sub-device can be provided.
- By Affinity Domain – Partition the device based on the affinity of the compute units to share the same level of cache hierarchy or to share a NUMA node. Sub-devices can be created from compute units that share the same:
    - NUMA node
    - L4 Cache
    - L3 Cache
    - L2 Cache
    - L1 Cache
    - Next partitionable affinity domain

"Next partitionable affinity domain" partitions the device along the next partitionable affinity domain, starting with NUMA first and then proceeding down to L4, L3, L2, and then finally L1 cache, finding the level in which the device can be further sub divided. For most NUMA platforms where the caches are integrated in the node, the next partitionable affinity domain is NUMA. For a non-NUMA platform, it would typically be the outermost cache level.

These options are controlled by the programmer through a list of properties provided as parameters in the Create Sub Devices call which is described in the next section.

The partitioning types supported by the OpenCL implementation can be queried (described later in this article).

Create a Sub-device

The Get Device ID call in OpenCL helps find an available OpenCL device in a platform. Once a device is found using the clGetDeviceIDs call, you can then create one or more sub-devices using the clCreateSubDevices call. This is normally completed after the selection of the device and before creating the OpenCL context.

The clCreateSubDevices call is:

```
cl_int clCreateSubDevices (
       cl_device_id in_device,
       const cl_device_partition_property *properties,
       cl_uint num_devices,
       cl_device_id *out_devices,
       cl_uint *num_devices_ret)
```

- *`in_device`*: The id of the device to be partitioned.

- `properties`: List of properties to specify how the device is to be partitioned. This is discussed below in more detail.
- `num_devices`: Number of sub-devices (used to size the memory for *out_devices*).
- `out_devices`: Buffer for the sub-devices created.
- `num_devices_ret`: Returns the number of sub-devices that device may be partitioned into according to the partitioning scheme specified in properties. If num_devices_ret is NULL, it is ignored.

Partition Properties

Understanding the partition properties is key for partitioning the device into sub-devices. After deciding the type of partitioning (Equally, By Counts, or By Affinity Domain), develop the list of properties to pass as a parameter in the clCreateSubDevices call. The property list begins with the type of partitioning to be used, followed by additional properties that further define the type of partitioning and other information, and then finally the list ends with a 0 value. Property list examples are shown in the next section that helps illustrate the concept.

The partition property that starts the property list is the type of partitioning:

- CL_DEVICE_PARTITION_EQUALLY
- CL_DEVICE_PARTITION_BY_COUNTS
- CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN

The next value in the list depends on the partition type:

- CL_DEVICE_PARTITION_EQUALLY is followed by N, the number of compute units for each sub-device. The device is partitioned into as many sub-devices as can be created that have N compute units in each sub-device.
- CL_DEVICE_PARTITION_BY_COUNTS is followed by a list of compute unit counts. For each number in the list, a sub-device is created with that many compute units. The list of compute unit counts is terminated by CL_DEVICE_PARTITION_BY_COUNTS_LIST_END.
- CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN is followed by the type of partitioning for the affinity, either NUMA, L4_CACHE, L3_CACHE, L2_CACHE, L1_CACHE, or Next Partitionable:
    - CL_DEVICE_AFFINITY_DOMAIN_NUMA
    - CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE
    - CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE
    - CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE
    - CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE
    - CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE

The last value in the property list is always 0.

Property List Examples

This section contains examples of property lists.

To illustrate this example, we have an example target machine as our device. The target machine is a NUMA platform with 2 processors, each with 4 cores. There are a total of 8 physical cores in the machine. Intel Hyper-Threading Technology is enabled. There are a total of 16 logical threads in the machine. Each processor has a shared L3 cache that all 4 cores share. Each core has private L1 and L2 caches. With Hyper-Threading Technology enabled, each core has two threads, so each L1 and L2 cache is shared between two threads. There is no L4 cache. See Figure 1.

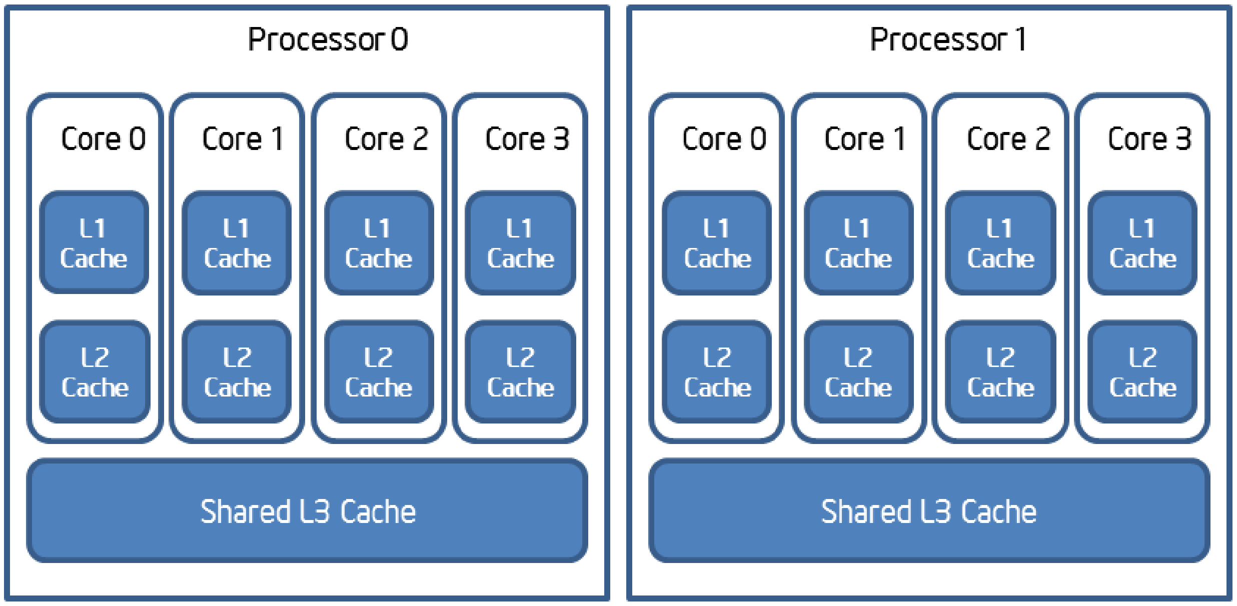

Figure 1. *Configuration of the Target Machine for Property List Examples*

The following table shows examples of property lists, assuming that the OpenCL implementation supports that particular partition type.

Notice the property lists always begin with the type of partitioning and end with a 0.

Table 1. *Property List Examples*

Property List	**Description**	**Result on the Example Target Machine**
{ CL_DEVICE_PARTITION_EQUALLY, 8, 0 }	Partition the device into as many sub-devices as possible, each with 8 compute units.	2 sub-devices, each with 8 threads.
{ CL_DEVICE_PARTITION_EQUALLY, 4, 0 }	Partition the device into as many sub-devices as possible, each with 4 compute units.	4 sub-devices, each with 4 threads.

{ CL_DEVICE_PARTITION_EQUALLY, 32, 0 }	Partition the device into as many sub-devices as possible, each with 32 compute units.	Error! 32 exceeds the CL_DEVICE_PARTITION_ MAX_COMPUTE_UNITS.
{ CL_DEVICE_PARTITION_BY_COUNTS, 3, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0 }	Partition the device into two sub-devices, one with 3 compute units and one with 1 compute unit.	1 sub-device with 3 threads and 1 sub-device with 1 thread.
{ CL_DEVICE_PARTITION_BY_COUNTS, 2, 2, 2, 2 CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0 }	Partition the device into four sub-devices, each with 2 compute units.	4 sub-devices, each with 2 threads.
{ CL_DEVICE_PARTITION_BY_COUNTS, 3, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0 }	Partition the device into two sub-devices, one with 3 compute units and one with 1 compute unit.	1 sub-device with 3 threads and 1 sub-device with 1 threads.
{ CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_NUMA, 0 }	Partition the device into sub-devices that share a NUMA node.	2 sub-devices with 8 threads each. Each sub-device is located on its own NUMA node.
{ CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE, 0 }	Partition the device into sub-devices that share an L1 cache.	8 sub-devices with 2 threads each. The L1 cache is not shared in our example machine.
{ CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE, 0 }	Partition the device into sub-devices that share an L2 cache.	8 sub-devices with 2 thread each. The L2 cache is not shared in our example machine.
{ CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE, 0 }	Partition the device into sub-devices that share an L3 cache.	2 sub-devices with 8 threads each. The L3 cache is shared among all 8 threads within each processor.
{ CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE, 0 }	Partition the device into sub-devices that share an L4 cache.	Error! There is no L4 cache.
{ CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE, 0 }	Partition the device based on the next partitionable domain. In this case, it is NUMA.	2 sub-devices with 8 threads each. Each sub-device is located on its own NUMA node.

Hyper-Threading Technology and Compute Units

If Hyper-Threading Technology is enabled, a computational unit is equivalent to a thread. Two threads share one core. If Hyper-Threading Technology is disabled, a computational unit is equivalent to a core. One thread executes on the core. Code should be written to handle either case.

Contexts for Sub-devices

Once the sub-devices are created, we can create contexts for them using the clCreateContext call. Note that if we use clCreateContextFromType to create a context from a given type of device, the context created does not reference any sub-devices that have been created from devices of that type.

Programs for Sub-devices

Just like creating a program for a device, a different program can be created for each sub-device. This is an efficient method to do task parallelism. Different programs can be created for different sub-devices.

An alternative is to share a program among devices and sub-devices. Program binaries can be shared among devices and sub-devices. A program binary built for one device can be used with all of the sub-devices created from that device. If there is no program binary for a sub-device, the parent program will be used.

Partitioning a Sub-device

Once a sub-device is created, it can be further partitioned by creating sub-devices from a sub-device. The relationship of devices forms a tree, with the original device as the root device at the top of the tree.

Each sub-device will have a parent device. The root device will not have a parent.

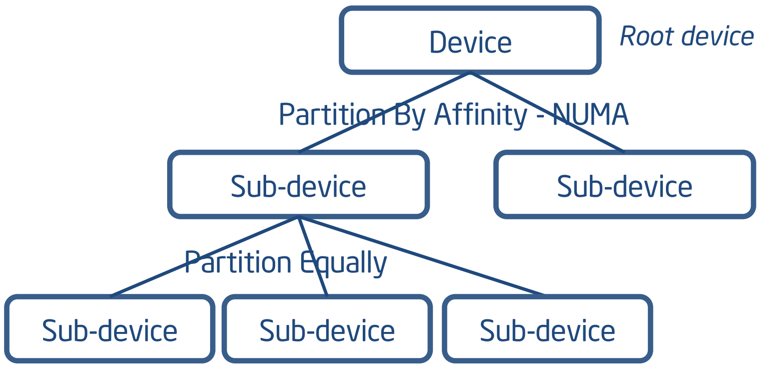

Figure 2. *Device Partitioning Example*

In Figure 2, we show an example of a device being partitioned first using Partition By Affinity Domain – NUMA, and then one of the sub-devices being partitioned using Partition Equally.

Query a Sub-device

The clGetDeviceInfo call has several additions to access sub-device related information.

Prior to creating sub-devices, we can query a device using clGetDeviceInfo to see:

- CL_DEVICE_PARTITION_MAX_SUB_DEVICES: Maximum number of sub-devices that can be created for this device.
- CL_DEVICE_PARTITION_PROPERTIES: Partition Types that are supported by this device.
- CL_DEVICE_PARTITION_AFFINITY_DOMAIN: List of supported affinity domains for partitioning the device using Partitioning By Affinity Domain.

Of course, we recommend checking that the Partition Type you want to use is supported. Some OpenCL implementations may not support all types.

After creating sub-devices, we can query sub-devices the same way devices are queried. Through querying, we can discover things like:

- CL_DEVICE_PARENT_DEVICE: Parent device for the given sub-device.
- CL_DEVICE_PARTITION_TYPE: Current partition type in use for this sub-device.

A query to a root device and all descending sub-devices should return the same values for almost all queries. For example, when queried, the root device and all descendant sub-devices should return the same CL_DEVICE_TYPE or CL_DEVICE_NAME. The exceptions are the following queries:

- CL_DEVICE_GLOBAL_MEM_CACHE_SIZE
- CL_DEVICE_BUILT_IN_KERNELS
- CL_DEVICE_PARENT_DEVICE
- CL_DEVICE_PARTITION_TYPE
- CL_DEVICE_REFERENCE_COUNT
- CL_DEVICE_MAX_COMPUTE_UNITS
- CL_DEVICE_MAX_SUB_DEVICES

Release and Retain Sub-device

There are two calls that allow the programmer to maintain the reference count of a sub-device. We can increment the reference count (retain) or decrement the reference count (release) just like other OpenCL objects. clRetainDevice increments the reference count for the given sub-device. clReleaseDevice decrements the reference count for the given sub-device.

Other Considerations

Here are some items to check for in your code when using device fission.

Check to see that device fission is supported for your device. Check the maximum number of sub-devices that can be created.

Check to see that the device fission partition type is supported. This can be checked using the GetDeviceInfo call.

After creating the sub-devices, check to see that devices are indeed created correctly. For example, if you are using Partition By Affinity Domain – L3 Cache, check to see if the expected number of sub-devices are created.

It is also important to make your code robust and able to handle future platform changes. Consider how your code will handle target hardware architecture changes in the future. Consider how the code will execute on a target machine with:

- New or different cache hierarchy
- NUMA or Non-NUMA platforms
- More or fewer compute units
- Heterogeneous compute nodes
- Hyper-Threading Technology enabled or disabled

Device Fission Code Examples

In this section, we show some simple code examples to demonstrate device fission.

Code Example #1 - Partition Equally

In this code example, we use Partition Equally to divide the device into as many sub-devices as possible, each with four computational units. (Error checking on OpenCL calls is omitted).

```
// Get Device ID from selected platform:

clGetDeviceIDs( platforms[platform], CL_DEVICE_TYPE_CPU, 1, &device_id, &numDevices);

// Create sub-device properties: Equally with 4 compute units each:

cl_device_partition_property props[3];
props[0] = CL_DEVICE_PARTITION_EQUALLY;  // Equally
props[1] = 4;                            // 4 compute units per sub-device
props[2] = 0;                            // End of the property list

cl_device_id subdevice_id[8];
cl_uint num_entries = 8;

// Create the sub-devices:

clCreateSubDevices(device_id, props, num_entries, subdevice_id, &numDevices);

// Create the context:

context = clCreateContext(cprops, 1, subdevice_id, NULL, NULL, &err);
```

Code Example #2 - Partition By Counts

In this code example, we partition the device by counts with one sub-device with 2 compute units and one sub-device with 4 compute units. (Error checking on OpenCL calls is omitted).

```
// Get Device ID from selected platform:
```

```
clGetDeviceIDs( platforms[platform], CL_DEVICE_TYPE_CPU, 1, &device_id, &numDevices);

// Create two sub-device properties: Partition By Counts

cl_device_partition_property_ props[5];
props[0] = CL_DEVICE_PARTITION_BY_COUNTS; // Equally
props[1] = 2;                             // 2 compute units
props[2] = 4;                             // 4 compute units
props[3] = CL_DEVICE_PARTITION_BY_COUNTS_LIST_END; // End Count list
props[4] = 0;                             // End of the property list

cl_device_id subdevice_id[2];
cl_uint num_entries = 2;

// Create the sub-devices:

clCreateSubDevices(device_id, props, num_entries, subdevice_id, &numDevices);

// Create the context:

context = clCreateContext(cprops, 1, subdevice_id, NULL, NULL, &err);
```

Code Example #3 - Partition By Affinity Domain (NUMA)

In this code example, we partition the device using Partition By Affinity Domain – NUMA. (Error checking on OpenCL calls is omitted).

```
// Get Device ID from selected platform:

clGetDeviceIDs( platforms[platform], CL_DEVICE_TYPE_CPU, 1, &device_id, &numDevices);

// Create sub-device properties: Partition By Affinity Domain - NUMA

cl_device_partition_property props[3];
props[0] = CL_DEVICE_PARTITION_BT_AFFINITY_DOMAIN; // By Affinity
props[1] = CL_DEVICE_AFFINITY_DOMAIN_NUMA;         // NUMA
props[2] = 0;                                      // End of the property list

cl_device_id subdevice_id[8];
cl_uint num_entries = 8;

// Create the sub-devices:

clCreateSubDevices(device_id, pprops, num_entries, subdevice_id, &numDevices);

// Create the context:

context = clCreateContext(cprops, 1, subdevice_id, NULL, NULL, &err);
```

Strategies for Using Device Fission

In this section, we discuss some different strategies for using device fission to improve the performance of OpenCL programs or to manage the compute resources efficiently. The strategies are not mutually exclusive as one or more strategies may be used together.

One pre-requisite to leveraging the strategies is to truly understand the characteristics of your workload and how it performs on the intended platform. The more you know about the workload, the better you will be able to take advantage of the platform.

Strategy #1: Create a High Priority Task

Device fission can be used to create a sub-device for a high priority task to execute on dedicated cores. To ensure that a high priority task has adequate resources to execute when it needs to, reserving one or more cores for that task makes sense. The idea is to keep other less critical tasks from interfering with the high priority task. The high priority task can take advantage of all of the cores' resources.

Strategy: Use Partition By Counts to create a sub-device with one or more cores and another sub-device with the remaining cores. The selected cores can be exclusively dedicated to the high-priority task running on that sub-device. Other lower priority tasks can be dispatched to the other sub-device.

Strategy #2: Leverage Shared Cache or Common NUMA Node

If the workload exhibits a high level of data sharing between work items in the program, then creating a sub-device where all of the compute units share a cache or are located within the same NUMA node can improve performance. Without device fission, there is no guarantee that the work items will share a cache or share the same NUMA node.

Strategy: Create sub-devices that share a common L3 cache or are co-located on the same NUMA node. Use Partition By Affinity to create a sub-device for sharing an L3 cache or NUMA node.

Strategy #3: Exploit Data Re-Use and Affinity

Without device fission, submitting work to a work queue may dispatch it to a previously unused or "cold" core. A "cold" core is one whose instruction and data caches and TLBs (cache for address translations) may not have any relevant data and instructions for the OpenCL program. It will take time for data and instructions to be brought into the core and placed into caches and TLBs. Normally this is not an issue, but this can be a problem if the code does not run for a significant period of time. By the time the program warms up the processor caches, the program may have reached its end. Typically, this is not critical for medium and long running programs. The time penalty for warming up the processor can be amortized across longer execution times and it is normally not an issue. For very short running programs, however, it can be an issue. In this case, we need to take advantage of warmed processors by ensuring that subsequent executions of a program are routed to the same processors as previously used. This can also arise when larger applications are created from many smaller programs. The program executing before the current one accesses the data and brings it into the processor. The subsequent program can take advantage of that work.

Strategy: Use Partition By Counts or Partition By Affinity to create a sub-device to specify specific cores for the work queue. Try to re-use the core's warm caches and TLBs, especially for short running programs.

Strategy #4: Enable Task Parallelism

For certain types of programs, device fission can provide an improved environment for enabling task parallelism. Support for task parallelism is inherent in OpenCL with the ability to create multiple work queues for a device. The ability to create sub-devices can take that model to an even higher level. Creating sub-devices each with their own work queue allows more sophisticated task parallelism and runtime control. Examples are applications that act like “flow graphs[1]” where dependencies among the various tasks that make up the application help determine program execution. The tasks within the program can be modeled like nodes in a graph. The node edges or connections to other nodes model the task dependencies. For complex dependencies, multiple work queues with multiple sub-devices allow tasks to be dispatched independently and can ensure that forward progress is made.

You can also create different sub-devices with different characteristics. The sub-device can be created while keeping in mind the types of tasks it will execute. There also may be cases where the host wants to or needs to balance the work across these work queues rather than leaving it to the OpenCL runtime.

Strategy: Enable task parallelism by creating a set of sub-devices using Partition By Affinity or Partition Equally. Create work queues for each sub-device. Dispatch work items to work queues. The host can then manage the work across multiple work queues.

Strategy #5: High Throughput

There may be cases where absolute throughput is important, but data sharing is not. Suppose we have high throughput jobs to execute on a multiprocessor NUMA platform but there is limited or no data sharing between the jobs. Each job needs maximum throughput, e.g., it can use all of the available resources like on-chip caches. In this case, we might get the best performance if the jobs were executed on different NUMA nodes. We want to ensure that the jobs are not executed on a single NUMA node and have to compete for resources.

Strategy: Use Partition By Affinity to create N sub-devices – one sub-device for each NUMA node. The sub-devices can then use all NUMA node’s resources including all of the available cache.

Conclusion

To summarize, device fission is an addition to the OpenCL specification that gives more power and control to the OpenCL programmer to manage which computational units execute OpenCL commands. By sub-dividing a device into one or more sub-devices, we can control where the OpenCL programs execute and if used carefully can provide better performance and use the available compute resources more efficiently.

The Device Fission extension for OpenCL 1.1 is available on the OpenCL CPU device supported by the Intel SDK for OpenCL Applications 2012. The SDK is available at www.intel.com/software/opencl.

---

[1] Intel® Threading Building Blocks Reference Manual – Chapter 6

APPENDIX: Device Fission in OpenCL 1.1

In OpenCL 1.1, device fission is available as an extension: OpenCL Extension #11 (cl_ext_device_fission), dated June 9, 2010. This section highlights most of the programming differences between the 1.1 extension and the OpenCL 1.2 specification.

It is recommended to generally follow the OpenCL 1.2 API for device fission as the 1.1 Extension may be deprecated in the future.

Include File

The include file for 1.1 Extensions, cl_ext.h, should be added to the code:

```
#include <CL/cl_ext.h>
```

Partition By Names

The 1.1 Extension supports an additional partition type not supported in OpenCL 1.2: Partition By Names. This allows the programmer to specify a list of compute unit names to partition the device. See the Extension document for more information.

Properties

The following table shows the equivalent properties for OpenCL 1.2 and the 1.1 Extension.

1.2 Property	**1.1 Extension Property**
CL_DEVICE_PARTITION_EQUALLY	CL_DEVICE_PARTITION_EQUALLY_EXT
CL_DEVICE_PARTITION_BY_COUNTS	CL_DEVICE_PARTITION_BY_COUNTS_EXT
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN	CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT
CL_DEVICE_AFFINITY_DOMAIN_NUMA	CL_AFFINITY_DOMAIN_NUMA_EXT
CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE	CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE
CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE	CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE
CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE	CL_AFFINITY_DOMAIN_L2_CACHE_EXT
CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE	CL_AFFINITY_DOMAIN_L1_CACHE_EXT
CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE	CL_AFFINITY_DOMAIN_NEXT_FISSIONABLE_EXT

The end of the partition property list is the list terminator: CL_PROPERTIES_LIST_END_EXT.

Please note most of the tokens above have different enumeration values between OpenCL 1.2 and the 1.1 Extension feature. The only exceptions are the list terminators.

clGetDeviceInfo Selectors

The table below lists matching cl_device_info values between the 1.1 Extension and OpenCL 1.2.

1.2 Selector	**1.1 Extension Selector**
CL_DEVICE_PARENT_DEVICE	CL_DEVICE_PARENT_DEVICE_EXT
CL_DEVICE_PARTITION_PROPERTIES	CL_DEVICE_PARTITION_TYPES_EXT
CL_DEVICE_PARTITION_AFFINITY_DOMAIN	CL_DEVICE_AFFINITY_DOMAINS_EXT
CL_DEVICE_REFERENCE_COUNT	CL_DEVICE_REFERENCE_COUNT_EXT
CL_DEVICE_PARTITION_TYPE	CL_DEVICE_PARTITION_STYLE_EXT

API Changes

The Create Sub Devices call in the 1.1 Extension is:

```
cl_int clCreateSubDevicesEXT(
    cl_device_id in_device,
    const cl_device_partition_property_ext * properties,
    cl_uint num_entries,
    cl_device_id *out_devices,
    cl_uint *num_devices );
```

Note that sizeof(cl_device_partition_property_ext) differs from sizeof(cl_device_partition_property) .

The Retain/Release Device API calls have an EXT suffix. They behave identically to their OpenCL 1.2 counterparts.

Behavior Changes

The 1.1 Extension does not support binary inheritance from the parent device. Binaries must be explicitly built for sub-devices.

The 1.1 Extension specifies that partitioning a device participating in a context created by clCreateContext causes the context to reference the resultant sub-devices. This behavior is not supported in the Intel 1.1 Extension implementation and was deprecated in the OpenCL 1.2 specification.

Notices