OpenCL™ Device Fission for CPU Performance

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

The Intel® SDK for OpenCL™ Applications is a comprehensive software development environment for OpenCL applications on Intel® architecture-based platforms. This SDK provides developers with the ability to develop and target OpenCL applications on Intel® CPUs using both the Windows* and Linux* operating systems.

The Intel SDK for OpenCL Applications 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 a feature in this SDK.

Download your FREE copy of the Intel SDK for OpenCL Applications at: [www.intel.com/software/opencl](http://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. As the number of computational units and threads have grown over time, it is useful to develop mechanisms to control these resources, rather than treating them as a single homogenous computing resource.

An important addition, named Device Fission, was made to the OpenCL specification to give OpenCL programmers control over which computational units execute OpenCL commands. Device fission was defined in the OpenCL 1.2 specification (and was previous 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 descendendent 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 requires 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 gives programmers greater control over the hardware platform by selecting which computational units are used by the OpenCL runtime to execute commands. The reason device fission 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:

- 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 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.

**How to Use Device Fission**

This section provides an overview on how to use device fission and create sub-devices in the Intel SDK for OpenCL Applications. Refer to section 4.3 (Partitioning a Device) of the OpenCL 1.2 specification for further details.
The different partitioning types and options available when creating sub-devices 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 Name** – Partition the device by compute units specified by device name. This is an Intel extension supported by the Intel SDK for OpenCL Applications. See OpenCL Extension #20, Version 2, August 15, 2013.

The partitioning types supported by the OpenCL implementation can be queried (described later in this article). Before you try to partition any device, it is highly recommended that you check your implementation to see what partitioning types are supported.

**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 OpenCL device and before creating the OpenCL context.

The clCreateSubDevices call is:

```c
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 a 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 Name), 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 to help 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_NAME_INTEL (Intel Extension)

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 NAME_INTEL is followed by a list of the compute unit names for this sub-device. The list of compute unit names is terminated by CL_DEVICE_PARTITION_BY_NAMES_LIST_END_INTEL.

The last value in the property list is always 0.

**Property List Examples**

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 (Intel® HT Technology) is enabled. The machine has a total of 16 logical threads. In this example, the logical threads of processor 0 are numbered by the OS as 0, 1, 2, 3, 4, 5, 6, and 7, and the logical threads of processor 1 are numbered as 8, 9, 10, 11, 12, 13, 14, and 15.

Each processor has a shared L3 cache that all 4 cores share. Each core has private L1 and L2 caches. With Intel HT Technology enabled, each core has 2 threads, so each L1 and L2 cache is shared between 2 threads. There is no L4 cache. See Figure 1.
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

<table>
<thead>
<tr>
<th>Property List</th>
<th>Description</th>
<th>Result on the Example Target Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>`{ CL_DEVICE_PARTITION_EQUALLY, 8, 0 }</td>
<td>Partition the device into as many sub-devices as possible, each with 8 compute units.</td>
<td>2 sub-devices, each with 8 threads.</td>
</tr>
<tr>
<td>`{ CL_DEVICE_PARTITION_EQUALLY, 4, 0 }</td>
<td>Partition the device into as many sub-devices as possible, each with 4 compute units.</td>
<td>4 sub-devices, each with 4 threads.</td>
</tr>
<tr>
<td>`{ CL_DEVICE_PARTITION_EQUALLY, 32, 0 }</td>
<td>Partition the device into as many sub-devices as possible, each with 32 compute units.</td>
<td>Error! 32 exceeds the CL_DEVICE_PARTITION_MAX_COMPUTE_UNITS.</td>
</tr>
<tr>
<td>`{ CL_DEVICE_PARTITION_BY_COUNTS, 3, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0 }</td>
<td>Partition the device into 2 sub-devices, 1 with 3 compute</td>
<td>1 sub-device with 3 threads and 1 sub-device with 1 thread.</td>
</tr>
<tr>
<td>{ CL_DEVICE_PARTITION_BY_COUNTS, 2, 2, 2, 2, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0 }</td>
<td>Partition the device into 4 sub-devices, each with 2 compute units.</td>
<td>4 sub-devices, each with 2 threads.</td>
</tr>
<tr>
<td>{ CL_DEVICE_PARTITION_BY_COUNTS, 3, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0 }</td>
<td>Partition the device into 2 sub-devices, 1 with 3 compute units and 1 with 1 compute unit.</td>
<td>1 sub-device with 3 threads and 1 sub-device with 1 threads.</td>
</tr>
<tr>
<td>{ CL_DEVICE_PARTITION_BY_NAMES_INTEL, 0, 1, 7, CL_DEVICE_PARTITION_BY_NAMES_LIST_END_INTEL, 0 }</td>
<td>Partition the device into 1 sub-device using these specific logical threads.</td>
<td>1 sub-device with 3 threads: Thread 0, Thread 1, and Thread 7.</td>
</tr>
<tr>
<td>{ CL_DEVICE_PARTITION_BY_NAMES_INTEL, 0, 8, CL_DEVICE_PARTITION_BY_NAMES_LIST_END_INTEL, 0 }</td>
<td>Partition the device into 1 sub-device using these specific logical threads.</td>
<td>1 sub-device with 2 threads: Thread 0 and Thread 8.</td>
</tr>
</tbody>
</table>

**Intel® HT Technology and Compute Units**

If Intel HT Technology is enabled, a computational unit is equivalent to a thread. Two threads share one core. If Intel HT 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, you can create contexts for them using the clCreateContext call. Note that if you 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

Figure 2 shows an example of a device being partitioned first using Partition By Counts, and then one of the sub-devices being partitioned using Partition Equally.

There may be restrictions in partitioning sub-devices. For example, a sub-device created using Partition By Names cannot be further sub-divided. A sub-device created with another partitioning type cannot be further sub-divided by Partitioning By Name.

Query a Sub-device

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

Prior to creating sub-devices, you 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.

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, you can query sub-devices the same way devices are queried. Through querying, you 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`
Release and Retain Sub-device

Two calls allow you to maintain the reference count of a sub-device. You 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

When using device fission you need to check that:

- device fission is supported for your device
- the maximum number of sub-devices that can be created is not exceeded
- 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.

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
- Intel HT Technology enabled or disabled

Device Fission Code Examples

This section shows 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 4 computational units. (Error checking on OpenCL calls is omitted).

```c
// 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
```
Code Example #2 - Partition By Counts

In this code example, we partition the device by counts with 1 sub-device with 2 compute units and 1 sub-device with 4 compute units. (Error checking on OpenCL calls is omitted).

```c
// 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);
```

Strategies for Using Device Fission

There are 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 prerequisite 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 Names 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 if the code does not run for a significant period of time. By the time the program warms up the processor caches, it may have reached its end. Typically, this is normally not an issue for medium and long running programs. The time penalty for warming up the processor can be amortized across longer execution times. For very short running programs, however, it can be an issue. In this case, you 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 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. With the ability to create sub-devices you 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. An example is applications that act like “flow graphs” 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 instead of leaving it to the OpenCL runtime.

*Strategy:* Enable task parallelism by creating a set of sub-devices using 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**
Sometimes absolute throughput is important, but data sharing is not. Suppose you 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, you might get the best performance if the jobs were executed on different NUMA nodes. This ensures that the jobs are not executed on a single NUMA node and have to compete for resources.

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

**Conclusion**
Device fission is a feature of the OpenCL specification that gives OpenCL programmers more power and control to manage which computational units execute OpenCL commands. By sub-dividing a device into one or more sub-devices, you 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 feature is available on the OpenCL CPU device supported by the Intel SDK for OpenCL Applications. The SDK is available at [www.intel.com/software/opencl](http://www.intel.com/software/opencl).

---

**About the Author**

**Terry Sych** is a Staff Software Engineer in the Platform Architecture Enabling group at Intel Corporation. He joined Intel in 1992, and has worked on performance analysis and software optimization of enterprise and cloud applications for the last 15 years. He received a BS degree in Computer Engineering from the University of Michigan in 1981 and an MSEE degree from the University of Minnesota in 1988. He holds 3 US patents.
Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel and the Intel logo are trademarks of Intel Corporation in the US and/or other countries.

OpenCL and the OpenCL logo are trademarks of Apple Inc and are used by permission by Khronos.

Copyright © 2014 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.