Multi-Device Basic

Sample User's Guide

Intel® SDK for OpenCL Applications - Samples*

Document Number: 329763-004US

Contents

Legal Information

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

About Multi-Device Basic Sample

The Multi-Device Basic sample is an example of utilizing the capabilities of a multi-device system. Such systems might have different hardware setups, for example:

- Systems based on CPU and GPU devices, where one OpenCL device is a regular CPU and another is an on-chip or a discrete GPU card.
- HPC systems based on CPU and discrete GPUs or accelerator devices like Intel® Xeon Phi™ coprocessors.

This sample targets systems with multiple Intel Xeon Phi coprocessor devices, but its guidelines and methods are also applicable to multi-device systems with CPU and GPU devices, or a CPU and one Intel Xeon Phi coprocessor device.

For optimal utilization of devices, you need to reduce idle time by loading multiple devices simultaneously.

The Multi-Device Basic sample exemplifies three basic scenarios of simultaneous utilization of multiple devices under the same system:

- System-level
- Multi-context
- Shared context

This sample demonstrates a minimal sequence of steps to keep all devices busy simultaneously. It consists of:

- A simple synthetic kernel, operating in 1-dimensional iteration space.
- A simple work partitioning strategy, which comprises dividing all work among devices equally, regardless of their compute capabilities.

The sample utilizes no data sharing and therefore no synchronization between the devices. This is a purely functional sample with no performance instrumentation and no performance reported as sample output.

Algorithm

The sample calculates a synthetic function, which is implemented in the kernel, on a pair of input buffers *a* and *b,* and puts the resulting values to the output buffer c. Each buffer consists of `work_size` elements. For each index `i` the sample calculates `c[i] = f(a[i], b[i])`,

where

- `i = 0..work_size-1`
- `f` is a function, implemented in the kernel.

Initial values for buffer elements are also synthetic: `a[i] = i, b[i] = 2*i`.

The aim of the sample is to demonstrate how to divide, allocate, share resources, and load several devices in the system simultaneously. The problem is simplified by excluding borders, halos, or other data shared between adjacent devices, which helps to omit overlapping between devices during resource partitioning.

The sample demonstrates basic steps to saturate multiple devices without complex explanations of access patterns and other issues. The Multi-Device Basic sample utilizes a static work partitioning approach instead of dynamic load balancing among devices. See the other SDK samples at software.intel.com/en-us/vcsource/tools/opencl.
In the considered scenarios the sample uses common math for work partitioning, which is suitable for the case where the number of data items is much larger than the number of devices.

According to this strategy, work is divided among devices evenly, and the non-dividable piece is assigned to the last device. For a case where you cannot divide data with small granularity, you need to utilize a different math to distribute the last piece of work among several devices for better load balance.

OpenCL* Implementation

The following scenarios are considered in this sample. Scenario names are not conventional. Each name is an alias of the respective scenario in the document and sample code.

- *System-level* scenario, where separate devices are picked up by different instances of the same application. Each instance gets its index and recognizes how many application instances run simultaneously to correctly divide work and process a specific portion.
- *Multi-context* scenario, where one application instance uses all devices, and each device has its own OpenCL* context.
- *Shared-context* scenario, where all devices are placed in the same shared context and share input and output buffers using sub-buffers.

Refer to the scenario-dedicated sections in this document to understand which scenario best suits your needs.

System-Level Scenario

In the system-level scenario, multi-device parallelism is implemented outside of the host application. Multiple instances of the sample application run simultaneously under the same system. The following illustrates the system-level scenario:

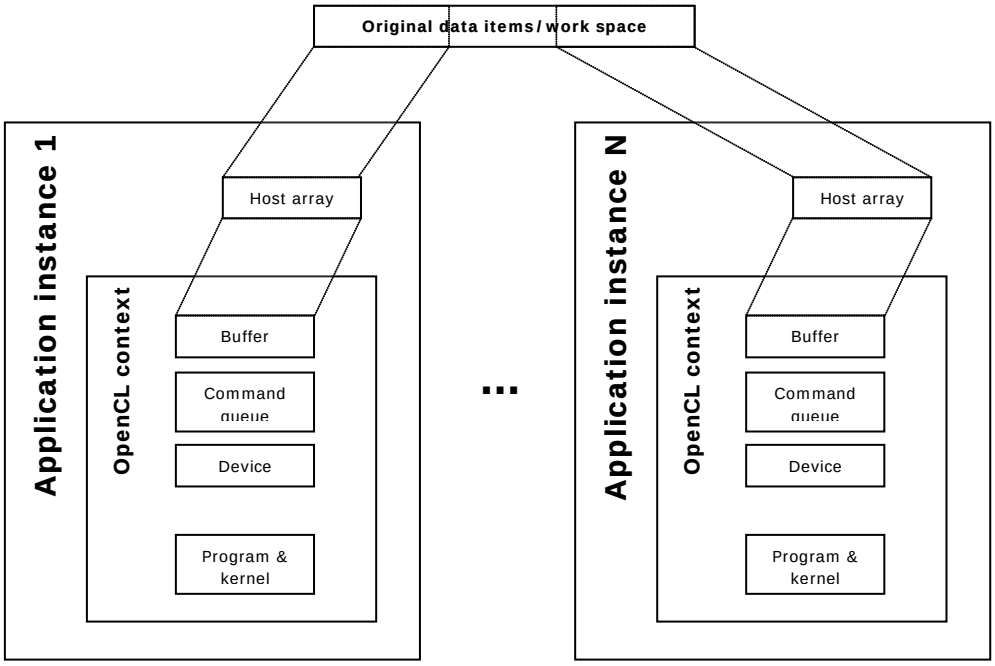

Fig. 1: System-level scenario

If you already have an OpenCL-enabled application with ability to partition work between multiple instances, for example, through MPI, then you do not need to modify this application to use multiple devices. Just run one instance per each device.

In case you have an MPI-enabled and OpenCL-enabled application (for a cluster) that is capable of utilizing one device, and you want to utilize a machine with multiple Intel® Xeon Phi™ coprocessors, you do not need to make any adjustments in work partitioning. Run one application instance per each Intel Xeon Phi coprocessor under the same system. Coprocessors have equal compute power, which means that the approach of dividing work between devices evenly provides the desired performance scalability, assuming that the execution time is distributed among work items uniformly.

Using the system-level scenario, you should limit the number of devices for each application instance externally, using one of the following methods:

- Setting the OpenCL device type with different values depending on application instance. In such a case, different application instances use different types of devices. For example, one instance uses CPU device, while another uses the coprocessor device. To set the device type, use the `-t` command-line option.

- Enabling the `OFFLOAD_DEVICES` environment variable on the systems with multiple Intel Xeon Phi coprocessors. `OFFLOAD_DEVICES` does not require any special processing by applications, as the environment variable is supported at OpenCL implementation level. `OFFLOAD_DEVICES` limits the Intel Xeon Phi coprocessor device visibility to a particular process in the system.

- Combination: set the device type and enable the `OFFLOAD_DEVICES` environment variable. In such case you can use a combination of CPU device and multiple coprocessors.

This sample implements the synthetic algorithm that does not involve any inter-device communication, so the code does not organize inter-instance interaction. Each application instance works individually and independently from others. Each instance calculates which work items to process so that all instances calculate the complete result but do not collect the resulting values into one place.

Multi-Context Scenario

In the multi-context scenario, one application instance uses all devices. Yet each device has its own context, so programs, kernels, buffers, and other resources are not shared. You should create the resources individually for each device. You can share only the memory allocation on the host. Each device exploits a separated piece of the allocated host memory by using its own buffer, created with `CL_MEM_USE_HOST_PTR`.

Individual paths for each of the devices start from almost very beginning, from the context creation. The same code is executed multiple times for different devices. So syntactically, the code consists of a number of loops over individual devices and queues.

Absence of tight synchronization between devices is a consequence of context separation. You cannot use events from one context in other contexts. So the synchronization that you can organize should involve host-side API calls. This sample explicitly waits for completion in all queues in loop over all devices with the `clFinish` call.

The following figure illustrates the multi-context scenario.

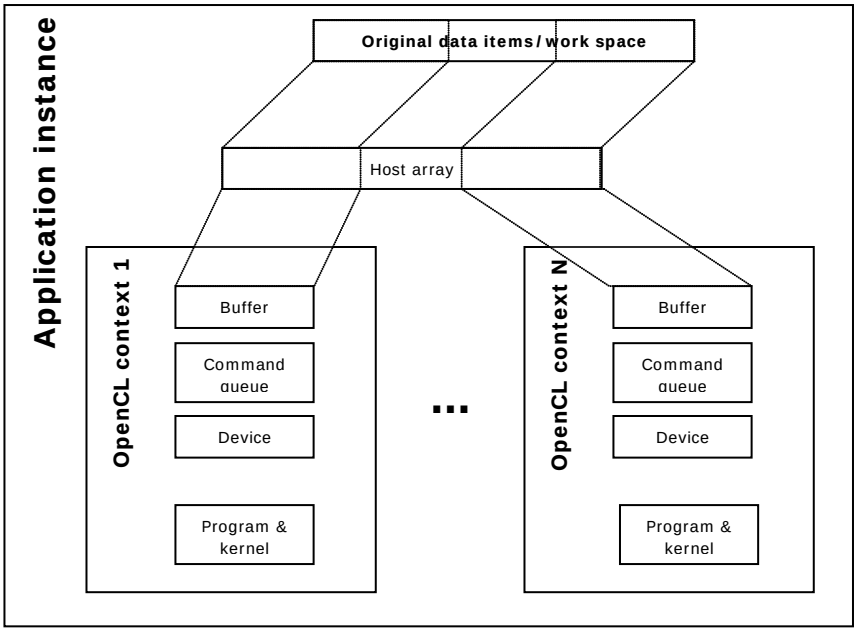

Fig. 2: Multi-context scenario

Shared-Context Scenario

In the shared-context scenario, program, kernel and all buffers are shared between all devices and exist in a single OpenCL* context. However, to use the output to the same buffer by multiple devices simultaneously, you need to create a non-overlapping sub-buffer for each device. See the OpenCL specification for more information.

The following figure illustrates the shared-context scenario:

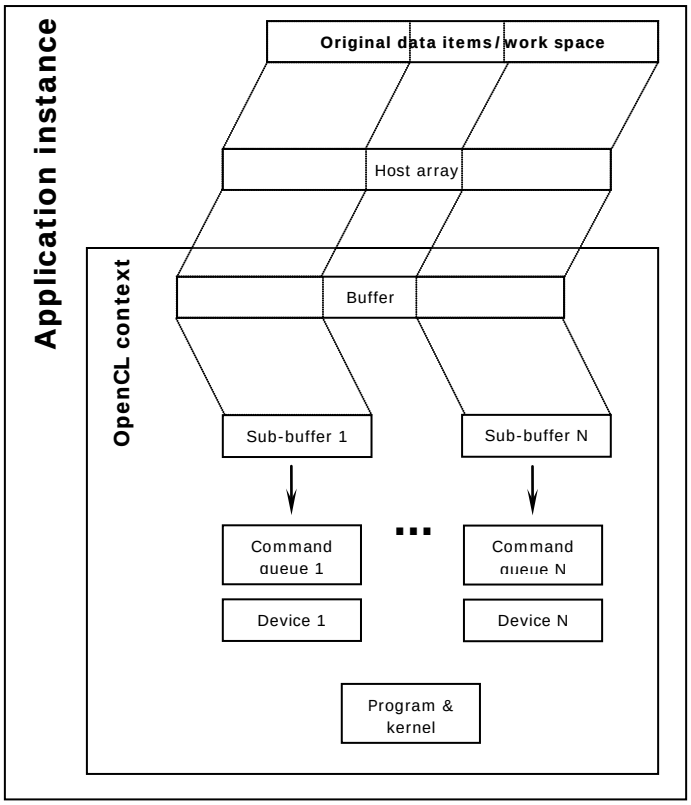

Fig. 3: Shared-context scenario scenscenario

You can also use OpenCL events to synchronize multiple devices without host-side participation, which is possible when all command queues coexist in a single context. Use this possibility to wait for the resulting buffer to become ready, which is the moment when all devices finish their NDRange commands. Specifically, use an array of event in the dependence list for the `clEnqueueMapBuffer` call. See more details in the source code.

To collect all devices of a specified type inside a single context, consider the following methods:

- Call `clGetDeviceIDs`, which lists the available devices. Call `clCreateContext` to create a context for the available devices.
- Call `clCreateContextFromType` directly for platform and device type. Call `clGetContextInfo` to query the available devices.

Querying the list of devices is necessary in all methods as you need to create a separate command queue for each device. OpenCL has no API for creating an array of command queues to simplify the process. This sample utilizes the method with calling `clCreateContextFromType`.

Choosing an Appropriate Scenario

You need to consider which scenario best suits your needs.

Prefer the system-level scenario if the inter-instance communication is not a bottleneck in your application. Otherwise, prefer multi- or shared-context scenarios, which provide more tight

synchronization capabilities between devices and hence better utilization of devices, particularly for application types that spend a lot of time for data transfers.

In the multi-context scenario, individual paths for each of the devices start from almost very beginning, from the context creation in comparison to shared-context scenario, which provides more sharing between devices (compare Fig. 2 with Fig. 3). Due to early separation, multi-context scenario has less flexibility, particularly in load balancing, and lack of tight synchronization between command queues, which requires more host participation in inter-device scheduling.

Using the multi-context scenario you can create a buffer for a dedicated device and avoid extra cycles for allocating (and potentially duplicating) a buffer in a context with multiple devices. This is relevant for the case when several Intel® Xeon Phi™ coprocessor devices are present in the OpenCL context, which requires some extra time to allocate the entire buffer and might not be suitable for the shared-context scenario.

Using the shared-context scenario you can organize efficient load balancing among devices by dynamically choosing sub-buffer sizes without recreating original buffers. Depending on the device type and OpenCL implementation, the cost of buffer creation might be higher than the cost of sub-buffer creation, so the dynamic load balancing approach can be efficiently implemented with shared-context.

Understanding the OpenCL* Performance Characteristics

Saturating Device Capabilities

You need to choose an appropriate work partitioning scenario to assign enough work to each device. Some types of devices, like Intel® Xeon Phi™ coprocessors, require a large number of work-groups that is scheduled in one NDRange. If the number of work-groups is insufficient, the system may result in device starvation and lead to lower performance on multi-device systems.

In multi-device context scenario, the given amount of work might be enough to utilize capabilities of one device, but not enough for several devices. Considering the overhead required for multi-device partitioning, dividing the work of one device into several devices can be slower than performing all work on one device.

Work-group Size Considerations

To provide each device with appropriate global size while dividing work between devices you should ensure enough granularity. The global size of NDRange enqueued for the device should be a multiple of a predefined value. You can query this value using `clGetKernelInfo` with `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE` for a particular kernel. The fixed granularity value satisfies minimal requirements for each pair of a device and a kernel in a multi-device environment, in case the value is used for all devices. This implies an additional requirement for the data partitioning scheme used in the application. While in the sample the additional requirement is not forced (to keep the source code shorter), in a real application the host logic should follow the requirement to achieve better performance results.

Generally this recommendation is implied by the auto-vectorization module of the compiler. See the Intel® SDK for OpenCL Applications - Optimization Guide for more information.

Project Structure

All files, necessary for sample build and execution, reside at the sample directory (`MultiDeviceBasic`) and in the `common` directory of the root directory, to which you extract samples.

`MultiDeviceBasic` directory contains the following files:

- Source files:

  - `multidevice.hpp` – declaration of main sample functions, which includes the sample scenarios and kernel creation function.
  - `multidevice.cpp` – entry point, command-line parameters definition, and parsing, selecting among scenarios and calling one of them.
  - `kernel.cpp` – creation of an OpenCL* program from a string; kernel code is inlined to this file.
  - `system.cpp` – implementation of the system-level scenario
  - `multi.cpp` – implementation of the multi-context scenario
  - `shared.cpp` – implementation of the shared-context scenario

- Scripts to run the system-level scenario with different hardware setups:
  - `cpu+mic.system-level.sh` – runs the system-level scenario with two application instances: one instance is for CPU OpenCL device, and another for the Intel® Xeon Phi™ coprocessor OpenCL device.
  - `multimic.system-level.sh` – runs the system-level scenario with several application instances, each instance is mapped for the dedicated Intel Xeon Phi coprocessor OpenCL device.
  - `cpu+multimic.system-level.sh` – runs the system-level scenario with CPU OpenCL device and with several application instances, each instance is mapped for a dedicated Intel Xeon Phi coprocessor OpenCL device.

NOTE: Multi-context and shared-context scenarios are executed directly by running the binary file with a specific command-line option without using any script files. Refer to the "Controlling the Sample" section for more information.

- Other files:
  - `Makefile` – builds the sample binary.
  - `README.TXT` – instruction on building and running the sample. Also provides information on understanding the sample output.

APIs Used

This sample uses the following OpenCL host functions:

- `clBuildProgram`
- `clCreateBuffer`
- `clCreateCommandQueue`
- `clCreateContext`
- `clCreateContextFromType`
- `clCreateKernel`
- `clCreateProgramWithSource`
- `clCreateSubBuffer`
- `clEnqueueMapBuffer`
- `clEnqueueNDRangeKernel`
- `clEnqueueUnmapMemObject`
- `clFinish`
- `clFlush`
- `clGetDeviceIDs`
- `clGetDeviceInfo`
- `clGetPlatformIDs`
- `clGetPlatformInfo`
- `clReleaseCommandQueue`
- `clReleaseContext`
- `clReleaseKernel`
- `clReleaseMemObject`
- `clReleaseProgram`
- `clSetKernelArg`
- `clWaitForEvents`

Controlling the Sample

You can run the following files in the command line:

- `multidevice`, the sample binary file, which is a console application.
- `<hardware_setup>.system-level.sh`, which is a script for running the system-level scenario in various hardware configurations, where `<hardware_setup>` is the placeholder for a hardware setup name.

The multi-context and the shared-context scenarios are executed directly by calling the sample binary with a particular `--context` command-line option. You can choose platform, devices, and other parameters through command line when calling the executable. To view all parameters, run the help command:

```
./multidevice –h
```

Help command shows the following help text:

Option	Description
`-h, --help`	Show this help text and exit.
`-p, --platform number-or-string`	Select platform, devices of which are used.
`-t, --type all | cpu | gpu | acc | default | <OpenCL constant for device type>`	Select the device by type on which the OpenCL kernel is executed.
`-c, --context system | multi | shared`	Type of the multi-device scenario used: with system-level partitioning, with multiple devices and multiple contexts for each device or one shared context for all devices. For one device in the system, system = multiple = shared.
`-s, --size <integer>`	Set input/output array size.
`--instance-count <integer>`	Applicable for system-level scenario only. Number of application instances which will participate in system-level scenario. To identify particular instance, use --instance-index key.
`--instance-index <integer>`	Applicable for system-level scenario only. Index of instance among all participating application instances which is set by --instance-count key.

Understanding the Sample Output

The following is an example of possible multidevice binary output for the shared context with CPU and Intel Xeon Phi coprocessor devices:

```
$ ./multidevice
Platforms (1):
[0] Intel(R) OpenCL [Selected]
Executing shared-context scenario.
Context was created successfully.
Program was created successfully.
Program was built successfully.
Number of devices in the context: 2.
Successfully created command queue for device 0.
Successfully created command queue for device 1.
Detected minimal alignment requirement suitable for all devices:
128 bytes.
Required memory amount for each buffer: 67108864 bytes.
Buffers were created successfully.
Sub-buffers for device 0 were created successfully.
Sub-buffers for device 1 were created successfully.
Kernel for device 0 was enqueued successfully.
Kernel for device 1 was enqueued successfully.
All devices finished execution.
```

First, the sample outputs all available platforms and picks one of them (line with [Selected]). Then it reports, which scenario is running. In the example, multidevice binary runs with no command-line parameters, so it executes according to the shared-context scenario by default.

Then sample reports each significant step of the OpenCL code execution and ends when all devices finish working.

Note that the sample reports no performance measures.

When running the system-level scripts, several multidevice binaries run at the same time. To avoid output mix and corruption, the output from each individual run forwards to a file. Output files have names, formed by name of the running script, and device type and number, particularly:

- For the cpu+mic.system-level.sh script:
  - cpu+mic.system-level.cpu.out -- for CPU device
  - cpu+mic.system-level.acc.out -- for Intel Xeon Phi coprocessor device
- For the multimic.system-level.sh script:
  - multimic.system-level.acc-I.out -- for I-th Intel Xeon Phi device, where I in {0..number of Intel Xeon Phi coprocessor devices minus one}
- For cpu+multimic.system-level.sh script:
  - cpu+multimic.system-level.cpu.out -- for CPU device
  - cpu+multimic.system-level.acc-I.out -- for I-th Intel Xeon Phi coprocessor device

References

Intel SDK for OpenCL Applications – Optimization Guide