**Lecture 14 - OpenCL programming**

**What is OpenCL?**

OpenCL is a standard API for accessing, especially, the GPU for doing general purpose computing in a program (not necessarily graphics related). However, the OpenCL can be used for accessing other specialized co-processors and even the CPU itself.

OpenCL is a standard developped by the Khronos group. It is similar to the CUDA, developped by NVIDIA for similar purposes; however, OpenCL is implemented on a wider variety of hardware.

**Concepts**

**Components**

The executable is linked against a (device-independent) library (OpenCL library).

At run-time, the OpenCL library must be able to find the driver(s) for the hardware on the execution machine.

**Concepts**

There is code that executes on the host machine, and code that executes on the OpenCL device (normally the GPU).

The code executing on the host can be written in any language; OpenCL provides a C API, but a C++ wrapper is available, and also bindings for other languages.

The code executing on the OpenCL device must be written in C, with some extensions and with some restrictions.

The main code (the code executing on the host) normally does the following things:

1. Queries the device capabilities;
2. Gives the code to be executed on the OpenCL device as a string to the OpenCL API and asks the OpenCL to compile the code;
3. Sends the data to the device;
4. Requests the execution of the device code;
5. Retrieves back the results.

**Work groups, work items, and memory types**

The code on the device is launched in multiple instances (somewhat like distinct processes or threads). The instances are called *work items*. Each work item can retrieve its work item ID and use it to decide which part of the work it is supposed to perform.

Work items are grouped in work groups. Within a work group, the work items can synchronize and can access the so-called *local* memory. No synchronization is possible among work items in different work groups.

The memory available to the code on the device is divided into:

* *global memory* — this is available to all the work items. It can also be copied from and to the host.
* *local memory* — there is an instance of it within each work group, and it is available to all work items in the same group.
* *private memory* — there is an instance of it within each work item, and it is available only to the owner work item.

The host API has an operation allowing to launch a specified number of work groups, each consisting in a specified number of work items.

**Execution queue and events**

The operations requested by the host code — memory transfer and work item executions — are placed in a queue. The host API allows to request enqueueing of such an operation.

To synchronize operations, each operation has an associated *event*, that gets signalled when the operation completes. It is possible:

* for an enqueued operation, to specify a list of events to be completed before the operation is allowed to start;
* in the host code, it is possible to wait for an event to be signaled.

**Simple examples**

* [opencl-sample1.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/opencl-sample1.cpp)
* [opencl-sample2.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/opencl-sample2.cpp)

**More complex cases — divide-and-conquer algorithms**

There are important limitations on what a kernel can do. Most importantly:

* A kernel cannot spawn another kernel; therefore, the host code must know beforehand how many instances of a kernel to launch.
* GPU code cannot be recursive.

Problem: how can we do the recursive split and combine steps?

Solution: do the splits and combines in stages, and make the host code aware of the number of necessary instances and make it drive the split operations and combine operations. See [opencl-bin-sum.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/opencl-bin-sum.cpp) and [opencl-mergesort.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/opencl-mergesort.cpp)

A diagram of a network

Description automatically generated