Subject: High Performance Computing **Sem: VI** **Department: CSE (DS and AIML)**

OpenCL Programming

The summary of steps to be performed for OpenCL Programming are given below:

1. Creating a context for an OpenCL program (selecting a device, such as a video card, CPU, or any available): *CLContextCreate(CL_USE_ANY)*. The function will return a context descriptor (an integer, let's denote it conditionally *ContextHandle*).
2. Creating an OpenCL program in the received context: it is compiled based on the source code in the OpenCL language using the *CLProgramCreate* function call, to which the text of the code is passed through the parameter *Source*:*CLProgramCreate(ContextHandle, Source, BuildLog)*. The function will return the program handle (integer *ProgramHandle*). It is important to note here that inside the source code of this program, there must be functions (at least one) marked with a special keyword ***__kernel*** (or simply *kernel*): they contain the parts of the algorithm to be parallelized (see example below). Of course, in order to simplify (decompose the source code), the programmer can divide the logical subtasks of the kernel function into other auxiliary functions and call them from the kernel: at the same time, there is no need to mark the auxiliary functions with the word *kernel*.
3. Registering a kernel to execute by the name of one of those functions that are marked in the code of the OpenCL program as kernel-forming: *CLKernelCreate(ProgramHandle, KernelName)*. Calling this function will return a handle to the kernel (an integer, let's say, *KernelHandle*). You can prepare many different functions in OpenCL code and register them as different kernels.
4. If necessary, creating buffers for data arrays passed by reference to the kernel and for returned values/arrays: *CLBufferCreate(ContextHandle, Size * sizeof(double), CL_MEM_READ_WRITE)*, etc. Buffers are also identified and managed with descriptors.

Class	Description
cl::Platform	Provides information about an OpenCL platform e.g. name, vendor, profile, and OpenCL extensions.
cl::Device	Represents an OpenCL device e.g. CPU, GPU, or other type of processor that implements OpenCL standard.
cl::Context	Represents a logical container for other classes. Users can start from context to query other information.

Class	Description
cl::CommandQueue	Represents a command queue which is a queue of commands that will be executed on an OpenCL device.
cl::Program	Represents a program which is a set of kernel functions that can be executed on an OpenCL Device. It provides methods for creating a program from kernel code, build a program, and provide ability to query for program information e.g. number of kernels, name, binary size, etc.
cl::Kernel	Represents an entry point of OpenCL function name to execute the entire kernel. Whenever users create a kernel, it needs a correct kernel function name as entry point to execute. Users can set arguments prior to execution.
cl::Buffer	Represents an OpenCL memory buffer which is a linear region of memory storing data for input and output from kernel execution.
cl::Event	Represents an OpenCL event in asynchronous manner for the status of OpenCL command. Users can use it to synchronize between operations between host and device.

Writing a First OpenCL Program

Let's see the steps for performing C = A+B in OpenCL.

The initial step is to add the OpenCL headers to the code.

Headers:

For the sake of greater readability, the examples are written using the C++ bindings of OpenCL (<CL/cl.hpp>).

```
#define CL_USE_DEPRECATED_OPENCL_2_0_APIS

#include <CL/cl.hpp>
```

Verifying the installed OpenCL platforms and setting up a device

One of the key features of OpenCL is its portability. So, for instance, there might be situations in which both the CPU and the GPU can run OpenCL code. Thus, a good practice is to verify the OpenCL platforms (cl::Platform) to choose on which the compiled code will run.

```
 cl::Platform default_platform = all_platforms[0];
  std::cout << "Using platform: " <<default_platform.getInfo<CL_PLATFORM_NAME>()
<< "\n";
```

An OpenCL platform might have several devices. The next step is to ensure that the code will run on the first device (device 0) of the platform, if found.

```
 std::vector<cl::Device> all_devices;
  default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);
  if (all_devices.size() == 0) {
      std::cout << " No devices found.\n";
      exit(1);
  }

  cl::Device default_device = all_devices[0];
  std::cout << "Using device: " << default_device.getInfo<CL_DEVICE_NAME>() << "\n";
```

The OpenCL Context

According to the OpenCL Parallel Programming Development Cookbook, "contexts are used by the OpenCL runtime for managing objects such as command queues (the object that allows you to send commands to the device), memory, program, and kernel objects, and for executing kernels on one or more devices specified in the context".

```
  cl::Context context({default_device});
```

Now we need to define the code object which is going to be executed on the device, called kernel.

```
cl::Program::Sources sources;
```

Allocating Memory on the Device

The host communicates with the device by using its global memory. However, it is required, first, to allocate, on the host portion of the code, to allocate memory on the device. In OpenCL, the memory allocated on the device is called Buffer.

In our example, we have to host vectors, A and B of SIZE=10.

```
int A_h[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int B_h[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
```

The next step is to allocate two regions on the device memory for A_h, B_h and C_h. It is a good practice to name the variables on the GPU with a suffix _d and _h on the host. This way, A_h is the version of the vector on the host and A_d its copy on the device.

```
cl::Buffer A_d(context, CL_MEM_READ_WRITE, sizeof(int) * SIZE);
```

In this example, buffer A_d is connected to the context, and it has the size of SIZE elements of four bytes (sizeof(int)).

A buffer can be of several types. In this tutorial, we focus on CL_MEM_READ_WRITE and CL_MEM_READ_ONLY. These flags say the actions will perform in the buffer.

In this tutorial, it is necessary to create three buffers: one CL_MEM_READ_ONLY for vector A, one CL_MEM_READ_ONLY for vector B, and a CL_MEM_WRITE_ONLY for vector C.

Creating a Queue

In OpenCL, it is required to create a queue to push commands onto the device. For those who program in CUDA, OpenCL queues are similar CUDA streams.

```
cl::CommandQueue queue(context, default_device);
```

Writing into the Device Memory

Once the CommandQueue queue is created, it is possible to execute commands on the device side.

Using the queue, connected to the context and the device default_device, it is possible to initialize the vectors A_d and B_d with the values from A_h and B_h.

```
queue.enqueueWriteBuffer(buffer_A, CL_TRUE, 0, sizeof(int) * SIZE, A_h);
queue.enqueueWriteBuffer(buffer_B, CL_TRUE, 0, sizeof(int) * SIZE, B_h);
```

Pay attention that the function is writing from the host to the buffer: enqueueWriteBuffer.

It is not required to initialize vector C as it will receive the values of A+B.

Building the OpenCL Kernel

In OpenCL, there are several ways to build the kernel function and enqueue its execution on the device. Unfortunately, to the best of our knowledge, these ways are lower level than CUDA, for which one just needs to define the block size, number of threads and call the kernel as a function.

In this tutorial, we present a way that programmers familiarized with CUDA might understand. In OpenCL, for the sake of higher portability, the kernel function is presented as a string, which is appended to the program in the runtime, as one can see in the code below.

The kernel simple_add is a substring of the global variable of type std::string kernel_code.

```
std::string kernel_code =
   "   void kernel simple_add(global const int* A, global const int* B, global int* C){ "
   "       C[get_global_id(0)]=A[get_global_id(0)]+B[get_global_id(0)];                 "
   "   }
```

In OpenCL, kernel functions must return void, and the global keyword means that the variable points to the global memory.

Organizing the source code for compilation

Initially, it is required to append the kernel, which is presented here as a string, to the OpenCL source code.

```
  cl::Program::Sources sources;
  sources.push_back({ kernel_code.c_str(),kernel_code.length() });
```

Then, we create an OpenCL program, linking the OpenCL source code to the context.

```
  cl::Program program(context, sources);
```

Then, the OpenCL code is ready to be compiled in execution time

```
if (program.build({ default_device }) != CL_SUCCESS) {
        std::cout << " Error building: " <<
program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(default_device) << "\n";
        exit(1);
   }
```

It is important to point out that compilation errors are found, in execution time at this point. If compilation errors are found, the program output the message Error building, and, then, outputs the compilation errors.

It is worth to mention that there are other ways to define a kernel in OpenCL. Furthermore, it is also possible to perform offline compilation of OpenCL kernels.

Launching the kernel on the device

From the program, which contains the simple_add kernel, create a kernel for execution with three cl:buffers as arguments.

```
    cl::make_kernel<cl::Buffer, cl::Buffer, cl::Buffer> simple_add(cl::Kernel(program,
"simple_add"));
    cl::NDRange global(SIZE);
    simple_add(cl::EnqueueArgs(queue, global), A_d, B_d, C_d).wait();
```

Next, it is possible to call simple_add as a function. However, note that in cl::EnqueueArgs(queue, global) the queue queue is passed and cl::NDRange global(SIZE) is the number of threads the execution is going to spawn on the device. The remaining arguments are the three buffers on the global memory of the device.

Kernel execution on the device

The kernel is a simple function that performs C[i]=A[i]+B[i]. In this case, OpenCL provides the function get_global_id(0), which returns the id i of the thread in a 1D organization.

Retrieving data from the device

In this example, it is only required to retrieve data from C_d to C_h.

```
queue.enqueueReadBuffer(C_d, CL_TRUE, 0, sizeof(int) * SIZE, C_h);
```

In the line above, we read, from the buffer C_d, sizeof(int) * SIZE bytes using the enqueueReadBuffer function.

Compiling an OpenCL code:

It is simple to compile an OpenCL code with gcc or g++. In short, it is only required to add -lOpenCL flag to the compilation command.

If it is not possible to find the OpenCL library, it is required, first, to locate libOpenCL and append the directory to the LD_LIBRARY_PATH or explicitly indicate the location of libOpenCL in the compilation commands.

```
g++ -lOpenCL exercise1.cpp
```

Program for implementing D=A+B+C

```
#define SIZE 10

#define CL_USE_DEPRECATED_OPENCL_2_0_APIS

#include <CL/cl.hpp>

#include <iostream>


using namespace std;



// kernel calculates for each element D=A+B+C

std::string kernel_code =

   "   void kernel simple_add(global const int* A, global const int* B, global int* C,global int* D){ "

   "       D[get_global_id(0)]=A[get_global_id(0)]+B[get_global_id(0)]+C[get_global_id(0)]; "

   "   }                                                                               ";

// kernel calculates for each element D=A+B+C

std::string other_kernel =

   "   void kernel other_add(global const int* A, global const int* B, global int* C,global int* D){ "
```

```
    "       D[get_global_id(0)]=A[get_global_id(0)]+B[get_global_id(0)]+C[get_global_id(0)];
"

    "   }                                                                              ";


int main() {


    //If there are no opencl platforms -  all_platforms == 0 and the program exits.


    //One of the key features of OpenCL is its portability. So, for instance, there might be
situations

    // in which both the CPU and the GPU can run OpenCL code. Thus,

    // a good practice is to verify the OpenCL platforms to choose on which the compiled code
run.


    std::vector<cl::Platform> all_platforms;

    cl::Platform::get(&all_platforms);

    if (all_platforms.size() == 0) {

        std::cout << " No OpenCL platforms found.\n";

        exit(1);

    }


    //We are going to use the platform of id == 0

    cl::Platform default_platform = all_platforms[0];

    std::cout << "Using platform: " <<default_platform.getInfo<CL_PLATFORM_NAME>()
<< "\n";




    //An OpenCL platform might have several devices.

    //The next step is to ensure that the code will run on the first device of the platform,
    //if found.

    std::vector<cl::Device> all_devices;
    default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);
    if (all_devices.size() == 0) {
        std::cout << " No devices found.\n";
        exit(1);
    }

    cl::Device default_device = all_devices[0];
    std::cout << "Using device: " << default_device.getInfo<CL_DEVICE_NAME>() << "\n";

    cl::Context context({ default_device });

    // create buffers on the device
    cl::Buffer A_d(context, CL_MEM_READ_ONLY, sizeof(int) * SIZE);
    cl::Buffer B_d(context, CL_MEM_READ_ONLY, sizeof(int) * SIZE);
    cl::Buffer C_d(context, CL_MEM_READ_ONLY, sizeof(int) * SIZE);
    cl::Buffer D_d(context, CL_MEM_WRITE_ONLY, sizeof(int) * SIZE);

    int A_h[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int B_h[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    int C_h[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    //create queue to push commands to the device.
    cl::CommandQueue queue(context, default_device);
```

```
//write arrays A and B to the device
queue.enqueueWriteBuffer(A_d, CL_TRUE, 0, sizeof(int) * SIZE, A_h);
queue.enqueueWriteBuffer(B_d, CL_TRUE, 0, sizeof(int) * SIZE, B_h);
queue.enqueueWriteBuffer(C_d, CL_TRUE, 0, sizeof(int) * SIZE, C_h);

cl::Program::Sources sources;

//Appending the kernel, which is presented here as a string.
sources.push_back({ kernel_code.c_str(),kernel_code.length() });
sources.push_back({ other_kernel.c_str(),other_kernel.length() });

//OpenCL compiles the kernel in runtime, that's the reason it is expressed as a string.
//There are also ways to compile the device-side code offline.
cl::Program program(context, sources);



if (program.build({ default_device }) != CL_SUCCESS) {
    std::cout << " Error building: " <<
program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(default_device) << "\n";
    exit(1);
}
//If runtime compilation are found they are presented in this point of the program.



//From the program, which contains the "simple_add" kernel, create a kernel for execution
//with three cl:buffers as parameters.
//The types must match the arguments of the kernel function.
```

```
    cl::make_kernel<cl::Buffer, cl::Buffer, cl::Buffer, cl::Buffer>
simple_add(cl::Kernel(program, "simple_add"));
    cl::make_kernel<cl::Buffer, cl::Buffer, cl::Buffer, cl::Buffer>
other_add(cl::Kernel(program, "other_add"));


    //Details to enqueue the kernel for execution.
    cl::NDRange global(SIZE);
    simple_add(cl::EnqueueArgs(queue, global), A_d, B_d, C_d,D_d).wait();


    int D_h[SIZE];
    //read result C_d from the device to array C_h
    queue.enqueueReadBuffer(D_d, CL_TRUE, 0, sizeof(int) * SIZE, D_h);


    std::cout << " result: \n";
    for (int i = 0; i<10; i++) {
        std::cout << D_h[i] << " ";
    }


    other_add(cl::EnqueueArgs(queue, global), A_d, B_d, C_d,D_d).wait();
    int other_D_h[SIZE];
    //read result C_d from the device to array C_h
    queue.enqueueReadBuffer(D_d, CL_TRUE, 0, sizeof(int) * SIZE, other_D_h);
    std::cout << " result: \n";
    for (int i = 0; i<10; i++) {
        std::cout << other_D_h[i] << " ";
    }
    return 0;
}
```