

GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 45
OpenCL - Runtime System (Contd.)

Hi, so let us welcome back to the lecture on GPU architectures and programming. So if you remember in the last lecture we have stopped midway in the kernel and buffer object creation commands that are available in OpenCL. Now so once we have got those definitions in place, so let us actually use them to create the read write buffers in which we can install a read from read from and write to where for the OpenCL kernels. So, these are will be creating a CL memory object.

(Refer Slide Time: 00:55)

```
OpenCL Host Code Cont.

Create memory objects

/* cl_mem clCreateBuffer(cl_context context, cl_mem_flags flags, size_t size,
    void *host_ptr, cl_int *errcode_ret) */

    1
// Create the input (a, b) arrays in device memory
cl_mem d_a = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, dataSize, h_a, &err);
cl_mem d_b = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, dataSize, h_b, &err);
// Create the output (c) array in device memory
cl_mem d_c = clCreateBuffer(context, CL_MEM_READ_WRITE, dataSize, NULL, &
    err);
```

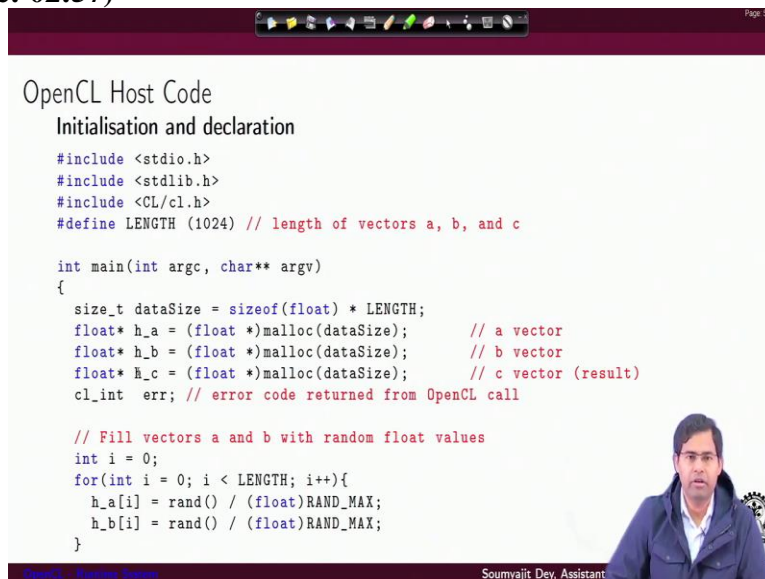
So you have a data. So, suppose you are creating the input arrays in the device memory, where you will transport data from the host program who can say host program. So, you execute the commands CL create buffer in that context and for the device and what you do is you have to specify the operation type. So, this is specified for that it can either to RCL memory read only or it CL MEM COPY HOST PTR.

So, this flags this memory flags actually tell that what are the operations? That are allowed in the buffer like you are allowed to copy data from the host side. So, with that you are creating these

two OpenCL memory objects `d a` and `d b` using the `CL create buffer` command inside the context. So, the buffers are created inside the context will be usable by the devices in the context you also create the output buffer where the system would be writing the output as you can see that again you create the `CL create buffer` in the same context.

But the memory flag is said `CL MEM READ WRITE`. Because now for this you also want that device side code to write its output OpenCL kernel should write this output. And the size of the buffer is given by this data size parameter using as you can see that size underscore `t` type argument is there and of course, they have to provide the host pointers that they are going to get information from which your host side code now, which host side memory objects, but if you remember, so just to recall things.

(Refer Slide Time: 02:57)



```
OpenCL Host Code
Initialisation and declaration

#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
#define LENGTH (1024) // length of vectors a, b, and c

int main(int argc, char** argv)
{
    size_t dataSize = sizeof(float) * LENGTH;
    float* h_a = (float *)malloc(dataSize); // a vector
    float* h_b = (float *)malloc(dataSize); // b vector
    float* h_c = (float *)malloc(dataSize); // c vector (result)
    cl_int err; // error code returned from OpenCL call

    // Fill vectors a and b with random float values
    int i = 0;
    for(int i = 0; i < LENGTH; i++){
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
    }
}
```

We have got this whole side array is, `h a`, `h b` and `h c` define. So the `h a` and `h b` are containing the input arguments, and `h c`, we plan to write back the data after their computation in the device said OpenCL code. So, here you have also specified that what are the operations to be done? And they have to be related with which host side data structures, `h a`, `h b` and all that. And then in this context, you have also created the buffer where you want the output back that is `d c`.


(Refer Slide Time: 03:38)

OpenCL Host Code Cont.

```
Set kernel arguments
/* cl_int clSetKernelArg(cl_kernel kernel,cl_uint arg_index,size_t arg_size,
  const void *arg_value) */

// Set the arguments to our compute kernel
err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
err = clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
err = clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
err = clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);

```



OpenCL - Basics Lecture Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, now the other thing that we will do is you want to pass arguments for the kernel. So, if you remember, do you have the rectorate kernel, they are available with you and this is the kernel object that we have created with the build kernel command earlier that we have discussed. Now, for this kernel object to our secreting this argument least by calling this function as many times as are the number of arguments.

So you want these to be provided with the 2 input buffer pointers, the output buffer pointer and the number of elements bound? So that is why you have 4 arguments. And you have these multiple calls. Now, the issue is why do we have multiple calls, it may happen that in each of these ideas, each of these initiatives of trying to provide arguments to this kernel, you face an error? So for each of those operations, you can have separate calls here.

So that in case you get errors in any one of them, it will be flagged out here. So I can write some handler code out of this that if in any of these operations, I get an error. If they do an order of this, then you exit from here.

(Refer Slide Time: 04:50)

OpenCL Host Code Cont.

Calculate global and local work size

```
/* cl_int clGetDeviceInfo(cl_device_id device, cl_device_info param_name,
    size_t param_value_size, void *param_value, size_t *param_value_size_ret
) */

const int count = LENGTH;
cl_uint max_work_itm_dims;
err = clGetDeviceInfo( device_id, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof
    (cl_uint), &max_work_itm_dims, NULL);
size_t *max_loc_size = (size_t*)malloc(max_work_itm_dims*sizeof(size_t));
err = clGetDeviceInfo( device_id, CL_DEVICE_MAX_WORK_ITEM_SIZES,
    max_work_itm_dims*sizeof(size_t), max_loc_size, NULL);
size_t global_work_size = count;
size_t local_work_size=1;
for (i=0;i<max_work_itm_dims;i++)
    local_work_size*=max_loc_size[i];
```



Now the issue is I want the kernel to work, but like a normal radar parallel program as we have seen that I need to pass it the work dimensions of the kernel what is the global work size and what is the local work size for this kernel. So, for this we have to identify still now whatever we have done is we have identified platforms or devices, but we need to identify what is the dimension, the kernel execution that a device can support.

Now, as you can see that this is something again which is very generic in OpenCL specific like inside a device, I can have support for multi-dimensional kernel and up to what dimensions specification of multi-dimensional kernel is allowed for the device may be device specific. So, the first we need to figure out what is the maximum work item dimension that is allowed in the specific device again, and just saying that despereaux vectorization this is not necessary.

We are trying to give you as much information system level information as possible. To this kind of code, so, for that we have this count variable, which is storing the length of the inputs on which you want to work. Now, here is the important function that will be using CL get device info, it is going to give back some CL type integers and you are passing it some device ID OpenCL device ID and you are passing it some parameter which you want to query that what is the parameter you want to query.

And then you did expect some values to be returned by this function and let us see the function working to get some more idea. So, you have also defined another you have also defined another integer here unsigned integer, which is which you wanted to store the maximum work item dimension, which is supported by the device? So Remember just a brief recall in CUDA you had grids and thread blocks, I mean grid containing thread blocks.

Here, you have a global work size and you say inside that you have work groups inside the work group you can have multidimensional arrangement of work items, you want to figure out in a given device, what is the maximum dimensional specification that is allowed for that for that device. So, for figuring that out, you have the max work item dimensions, you make a call to CL get device info for the specific device ID you pass it this instruction that may, you have to return to the maximum work item dimension you support.

So, you give this flag here, CL device max work item dimensions and you want that value to be returned. And so, of course, the other thing you do is you provide here as high specification of the parameter value and that would be a type that so, you have a size_t specification here which is like the size of what is the size of a uint unsigned integer of OpenCL. So, here you are calculating that size by size of operations and C and that gives you a value of type size_t.

And that is something you are passing here because you want the parameter values size should be known that in perform the function will give you back the parameter value size because that is also something that may be device specific. And so, once that is given, you are also passing it a pair of pointer here, which is going to stand that value. Now observe the thing, this pointer is for data of types cl_uint. So, you have to pass the pointer here and you have to also pass the size of the datatype for which the pointer will be initialized.

So this is a plain and simple design unsigned integer type pointer, you want to give these to the these are low level get device phase of quite low level function it needs to know from the runtime that. When I returned back some value what should be the maximum allow length for the value. So, since it is real type on size_t I calculate using size of what is the number of bytes

allowed for that. So that I can get to know in what size I will return with value and the last parameter returns null for this call will see it working later on.

So, with this call, what do we get is the maximum work item dimension parameter initialized like this for this device this is the maximum dimension that is allowed. Now, once that dimension information is provided to you, what you do is you create you are just creating a pointer here max local size and you are locking it with dynamically for each location size of size underscore T and what do you really want to understand that in each dimension first of all I have already figured out what is the maximum work item dimension, what is the next information? I mean, I would like to know.

In each dimension, what is the maximum value that is allowed like they work items so, suppose the work item dimension is 3 so, we know that I can have work items packed with entities of I mean x, y and z dimensions or less as a 1201 and 2 dimensions. Now, I want to know, what is the maximum value I can have in this dimension? What is maximum I can have in this dimension to know that I create this max loc size array?

And the important thing is for these area how many positions do I need for storage, or you need these many positions each have size underscore t. And finally, it will return back a pointer pointing to this size of n size of t will of course, so, there is a size of here and for the malloc call, you are not providing it the parameters that that is you give me an array containing space enough to hold data of this type and for these many dimensions.

So, with this since, let us say I am going coming back to the example, if the max dimension is 3, essentially I am going to initialize an array, which can work 2 values? And that is what this call is doing. Why do we want to do that? Because again, so you can see the pattern here. Again, I am going to make a call to CL get device info for the device ID. But now I provide it with a different parameter. Earlier, I queried it with network item dimension.

Now I am quieting it with maximum work item sizes. Of course, the implicit thing is in each dimension that is why now the return will not be to a single pointer, but now the return to be an

error? I mean, it is not the return value was not returned to a single point earlier to a single position. Now it has to be a set of values that needs to be written. So that is why the return type has not changed as you can see.

You are expecting this thing to be filled up this max local size you are expecting this to be filled up. So earlier, it give you back this parameter value size. Now, you are expecting it to so, earlier you give it you it just gives you the maximum dimension. Now, you have given it an array for storing the maximum dimensions in each of the maximum values possible in each of the dimensions. So, for that you have max local size which has been initialized and past year.

And here you have given it the size of this array of course, in the C type which will function. So, you just not just do not give the pointer here you also give me the maximum size which you already know by the way from the max marketing dimension multiplied by the size of by of the silence value t. So, with this, you expect that the runtime system will now contain in different positions of max loc size 0 max loc size 1 and max loc size 2 like that it will contain the maximum value of a local thread ids in each of those dimensions.

So, once this is known, you will like to figure out your global works, I mean, you like to figure out how many what is your local work size and all that. So, of course, the global work size is count because that many values are there to be operated one. Now, how does your local work size the all you simply want to do is you have to just identify what are the maximum work item dimensions in each of the mean in each of the time work item dimensions what is the maximum value which is already stored in different arguments of this array and you just loop over the array and multiply the values to get the local work size.

(Refer Slide Time: 14:44)

OpenCL Host Code Cont.


Writing input data to device from host

```

// cl_int clEnqueueWriteBuffer(cl_command_queue command_queue,cl_mem buffer,
cl_bool blocking_write,size_t offset,size_t cb,const void *ptr,cl_uint
num_events_in_wait_list,const cl_event *event_wait_list,cl_event *event)

//Write the data from host to the compute device
err = clEnqueueReadBuffer( commands, d_a, CL_TRUE, 0, sizeof(float) * count,
h_a, 0, NULL, NULL );
err = clEnqueueReadBuffer( commands, d_b, CL_TRUE, 0, sizeof(float) * count,
h_b, 0, NULL, NULL );

```



OpenCL - Runtime System Soumyajit Dey, Ass

So, once this is done, that is you have the local work size figured out. The next thing is you are going to write input data to the device from opposed now, the real game of executing the candidate begin. So, for that, you have to start you have to again you understand that an OpenCL everything is through the command queues use you give commands to the Enqueue commands to the commands queues, those command are going to be executed by the runtime system.

So you have to Enqueue commands for reads and write. The first thing would be that you want to Enqueue the host side values to the device side buffers in the context that you have set up. So near context, that you have the command queue command that you have already set up in this command, queue, you are Enqueueing, read buffer commands, to read from the host side, array h a to the buffers that you have defined inside your context, which are d a.

So just being back this way or declarations of buffers memory objects inside your context of execute. And they have already been told that to I mean, what are the host pointers from where they are which they are going to read? Now you just include the read operations here. So you went to read operations for the 2 arrays d a and d b.

(Refer Slide Time: 16:15)


OpenCL Host Code Cont.

Executing the kernel on device

```
//cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,cl_kernel
kernel,cl_uint work_dim,const size_t *global_work_offset,const size_t *
global_work_size,const size_t *local_work_size,cl_uint
num_events_in_wait_list,const cl_event *event_wait_list,cl_event *event)

// Execute the kernel over the entire range of our id input data set
err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, & global_work_size
, &local_work_size, 0, NULL, NULL);

//global_work_size (local_work_size) point to an array of work_dim unsigned
values that describe the number of global (local) work-items in work_dim
dimensions that will execute the kernel function.
```



OpenCL - Basics Lecture Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Once this is done, the next thing would be that you have to execute the kernel. So again, you have to Enqueue the execution command for the kernel. Using `clEnqueueNDRangeKernel`, there is the Enqueueing gives these execution command in the command queue for the kernel `ko_vadd` does a kernel object we have created earlier. And now you provide it with the global and local work size, global workspace was already known.


We needed that generic code to figure out what is the local work size allowed for this device? Because things in it can vary across devices. So in this case, for this dynamic, using this dynamic is of course. We have figured out what is the work item dimension and then we figured out in each of the dimensions what is the maximum work item size, we are multiplied them to get the local work size.

So, we have provided these global and local work sizes and now the kernel we expect the kernel execution command to be executed by the command queue to be launched to the device and getting executed. So, these global work size and local work size they point to an array of work dim, unsigned values that describe the number of work items in the work dimension that will execute in the kernel function will see examples of this.

(Refer Slide Time: 17:39)

Work-pool

- ▶ The work-groups associated with kernel-instance are placed into a logical pool of "ready to execute" work-groups called work-pool.
- ▶ OpenCL does not constrain the order how work-groups are scheduled for execution in the actual device from the work-pool
- ▶ Once in the work-pool, independent execution of work-groups can happen in any order and could be interleaved
- ▶ For each device there can be only one work-pool used by all command-queues associated with that device



OpenCL - Runtime System Soumyajit Dey, Assistant

So next comes the some idea about the work groups. So essentially, the work group that you have. So in OpenCL when you have launched a kernel now with the global and local work size information, the work groups associated with the carnal instance are placed into a logical pool of ready to execute work groups called work pool. Again, I will just repeat one thing just in case you are wondering why we did all this thing to figure out the local work size essentially we figured out what is the maximum number of what is the maximum work group size that is supported by the device.

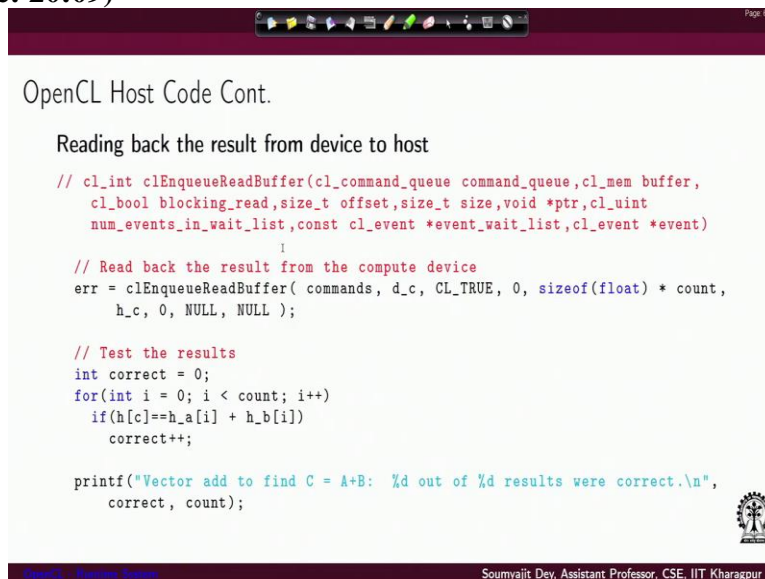
If we pass it here, then we are expecting that the devices will execute very efficiently it will pack as many threads as possible in the local work groups and execute them. So, that gives you the most efficient execution. So, this work pool is essentially the ready to execute work groups of course have to understand that just like in CUDA, you can have, you can launch multiple thread blocks, different thread blocks can be at any at any given time point in different states of execution like that in OpenCL you have work groups associated with the kernel instance.

And you are essentially creating a logical pool of ready to execute work groups and they are called work pool. Now OpenCL does not constrain the order on how work groups are scheduled for execution in the actual device from the work pool. So there is just like CUDA, you live it to the onboard scheduler and runtime system to manage this execution of workflows. So, once in

the work pool, you have independent execution of work groups, which can happen in any order and they can be interleaved among each other.

So, some work pool execute up to some point and then maybe they are waiting for some long latency operation to complete and by that time some other work from other work or pool can be launched and they can be executed and so on, so forth. For each device, there can be only one work pool used by all command queues associated with the device. So this is important for each device in the OpenCL runtime system, you can have only 1 work pool, which is used by all the command queues associated with the device. There cannot be more than that many that work. I mean, you can have multiple work pools which are used by this command queues which are associated there.

(Refer Slide Time: 20:09)



```
OpenCL Host Code Cont.

Reading back the result from device to host

// cl_int clEnqueueReadBuffer(cl_command_queue command_queue, cl_mem buffer,
// cl_bool blocking_read, size_t offset, size_t size, void *ptr, cl_uint
// num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)
1
// Read back the result from the compute device
err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0, sizeof(float) * count,
h_c, 0, NULL, NULL );

// Test the results
int correct = 0;
for(int i = 0; i < count; i++)
    if(h[c]==h_a[i] + h_b[i])
        correct++;

printf("Vector add to find C = A+B: %d out of %d results were correct.\n",
correct, count);
```

OpenCL - Runtime System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

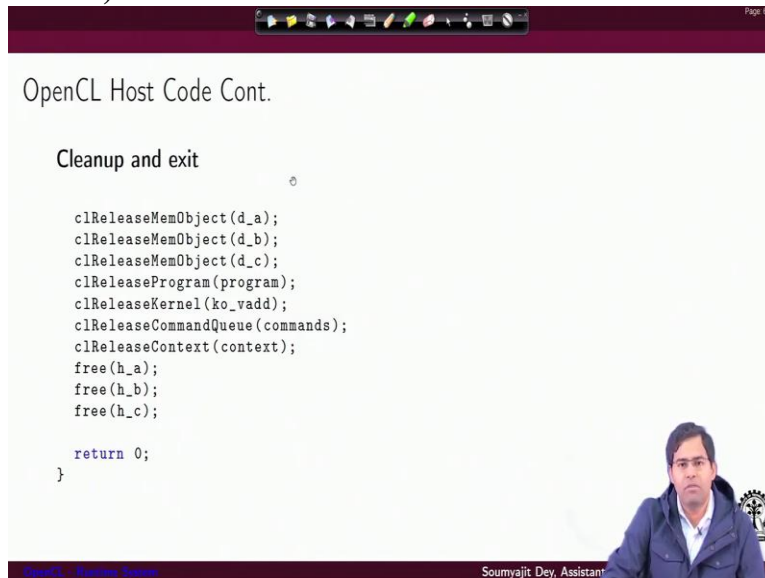
So now let us come back to the execution of the host program that we are discussing. So once we have into the range the kernel so now we are executed so that kernel I mean a first execution we did like it to be a to read back the values that have been actually written by the kernel. So, something we are expecting that the kernel is going to read the specification of this kernel is that is going to read its input arguments, because we have if you remember, we have already provided the arguments.

So it knows what are it is into I mean, what is the location from where it is going to read and of course, by the definition, the kernel is going to back in the d c memory buffer. So, at this point,

we know that after the kernels execution for including a read buffer commands, so, once the conversation completes, I can get the data in d c transferred back to h c. So, that read buffer command is executing now include in the command queue.

So, again, you will see that read command is also you do not directly read you execute Enqueue the read command in the command queue. Here some small test code here for testing whether the values are really correct or not after the communication and go for execution is open cell runtime. So you can so this is something as simple host size code, it will execute in the host size and you can just compare whether the h c value that have been computed by the device size code, they compare with the host size additions.

(Refer Slide Time: 21:49)



```
OpenCL Host Code Cont.

Cleanup and exit

clReleaseMemObject(d_a);
clReleaseMemObject(d_b);
clReleaseMemObject(d_c);
clReleaseProgram(program);
clReleaseKernel(ko_vadd);
clReleaseCommandQueue(commands);
clReleaseContext(context);
free(h_a);
free(h_b);
free(h_c);

return 0;
}
```

Now once this is done, as we can see that in you have I mean you are done with your main computation. So before for leaving the system you did like to free all the objects that we have created. So, you have CL release name object command to create to actually release all the memory objects, you have CL release program to release the program object and you can also release the kernel object can release the command queue.

And finally, so, this has to be done in this sequence you remove memory objects then you do program object you know kernel once is that done and the command queue is also released, then you remove the final context you release the final context. And also, you release the host side

arrays h a, h b, h c system becomes free from all the defined and allocated data and the new return back gracefully from the cont.

So, as you can see that these are quite complex OpenCL host code, the complexity primarily comes from the way we are viewing a complex architecture by an himself first of all assuming that the architecture is complex and we have to execute a lot of commands to create the context of execution, we can select, which are the devices from different vendors that I can pack in the context and all that.

But we have to understand that most of this code is I mean, is it going to be fixed? If you are going to use that same device, same platform, same GPU or same CPU for programming with some other execution? You just need to change the maybe you are definitions of buffers your definition of kernel and all that setting of platform setting up context and everything can remain same, unless and also if we are going to use that device again, maybe you do not even need to change the command queue and all those definitions.

So the point I am trying to make is most of the code of for the host side is going to remain same, if you are going to write another functionality. Write, another set of kernels are orchestrated execution of another set of kernels in the same platform defies command you write so most of the code is going to be same as a developer, only thing you need to do is modify the other parts of the code.

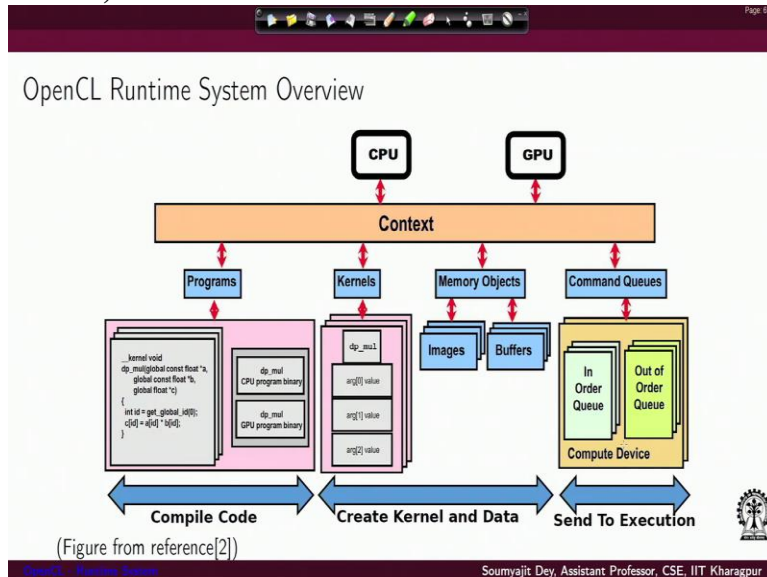
So that brings the question of this CUDA versus OpenCL issue with all these things and why do OpenCL at all? Well, let us remember that CUDA programs will execute only on your open. So if I mean Nvidia type GPUs, but OpenCL code will execute in any system where your devices are compliant with the khronos specified OpenCL specification. It can be a collection of CPUs even it can be Nvidia GPUs, it can be opens I mean AMD GPUs, Intel CPUs, AMD CPUs somebody else is devices.

Some other accelerators which support in all these OpenCL specification and all that. And as we know that Xilinx based FPGAs also support OpenCL. So it can be a heterogeneous system

comprising multiple such devices. So there is the advantage so you do not get specific to one vendor. And but that also makes it much more system level programming. Of course, if you are using OpenCL bindings like C++ extensions or the few OpenCL the Python extensions of OpenCL.

Then maybe in your program all these low level details you do not need to do you can write a much simpler OpenCL program. With this we have the specification of the host program given here at the lowest level that is a when as sea level OpenCL program just to provide you with the more intricate device specific details of an actual OpenCL problems execution.

(Refer Slide Time: 25:46)



(Figure from reference[2])

So, coming to my summary that we like to make here. So what really is the runtime systems overview what are the different phases that you have gone through so we are trying to show a simple example here. You have a CPU and GPU they are, these are the 2 devices, let us say you have merited them inside a context, we know how to do it. Now, you have this programs, the programs sources.

The kernel sources, and you can have from the kernel source, you can compile them to create CPU binaries you can compile them to create GPU binaries using suitable different compilation systems or you can actually even use our methods to create kernel programs from the sources? And then you can actually use our methods from using the program objects to create the kernel objects, then, so that is there is a sequence.

You have sources from then you can create program objects from them, you can create kernel objects, then you parameterize the kernel with the different arguments here. So using the set kernel argument commands, you can specify the memory objects that are to be so, all these things go inside the context of execution this context means everything is here every execution orchestration will be happening inside this abstraction.

why do we again we will repeat why do we define this abstraction technically we are viewing a more complex system there can be other CPUs, GPUs encapsulated inside another context type object to bring in this abstraction we can have multiple context. So, inside this context we can have program objects kernel objects and we can also have memory type objects of course, they are required.

So, that would be normal memory buffers OpenCL also provides insight into internal support for an image type memory objects which are already there. So, with this you have the data read write parts ready, where from to read what kind of data to read and where to write, you have the execution kernels steady they are created from a program objects. And then the important thing is, as we are saying that you do not really launch the commands the runtime systems are going to launch the commands.

But the order in which things would happen you can specify using the command queue that will you read write execute in this order. And when you execute those commands, you can actually give instruction whether execute them in order or whether it be executed out of order like that. So, these are the different phases through which a typical OpenCL programs execution would go through. With this will be ending this lecture. Thank you for your attention.