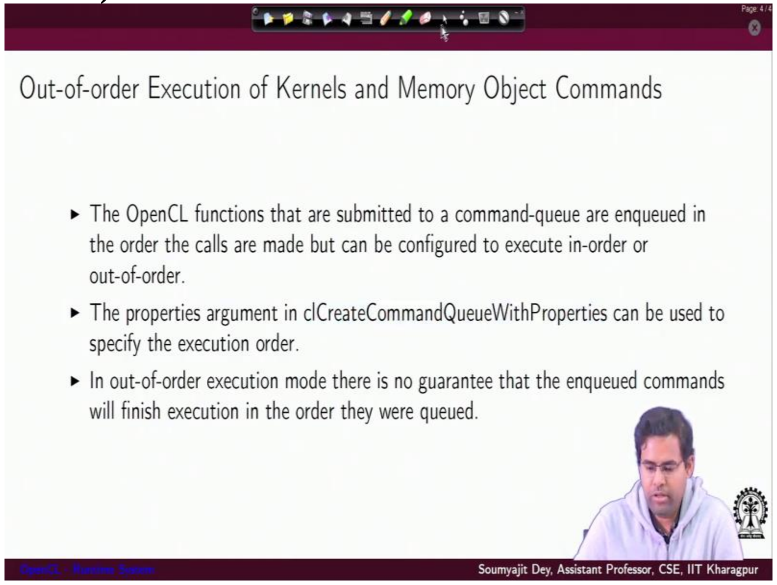
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 47
OpenCL - Runtime System (Contd.)

Hi, so, in the previous lecture on a GPU programming and architecture we have been discussing on in order execution of commands and different synchronization primitives. For example, we discussed how work groups can be synchronized, how you can put in barriers inside a command queue. And then there was this idea of how you can use CL wait for events primitive because with every incoming command you have an event return that event will have different status messages.

And this status you can make you can have a CL works for event function using which you can query the event status and unless a specific event reaches a specific status, you can also wait at those points. And you also have these primitives like CL finish and CL flush, which are also useful from that perspective. And now, the next thing we want to discuss was that well even if you commands in a command queue, we usually assume that.

(Refer Slide Time: 01:22)



Out-of-order Execution of Kernels and Memory Object Commands

- The OpenCL functions that are submitted to a command-queue are enqueued in the order the calls are made but can be configured to execute in-order or out-of-order.
- The properties argument in clCreateCommandQueueWithProperties can be used to specify the execution order.
- In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued.

OpenCL - Runtime System

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

The commands will be issued to the underlying device for which the queue has been made in an in order fashion, which is not true, it will also happen that you can if you want you can execute the commands out of order and you let the runtime system decide exactly in which order the commands will be executed. So, just to give an example, what we mean suppose this is the command queue, and you have executive and you have been queued, read command, then a kernel lunch point kernel has been include.

And then, you have let me just redraw this figure suppose, I am trying this graphically. So, you have cl command queue write buffer followed by kernel branch, then you have a read buffer command after this, then you launch another kernel k prime like that. So, this if you have a normal command queue, which is to be executing in order, this is exactly the order in which the different commands will be launched to the underlying device right.

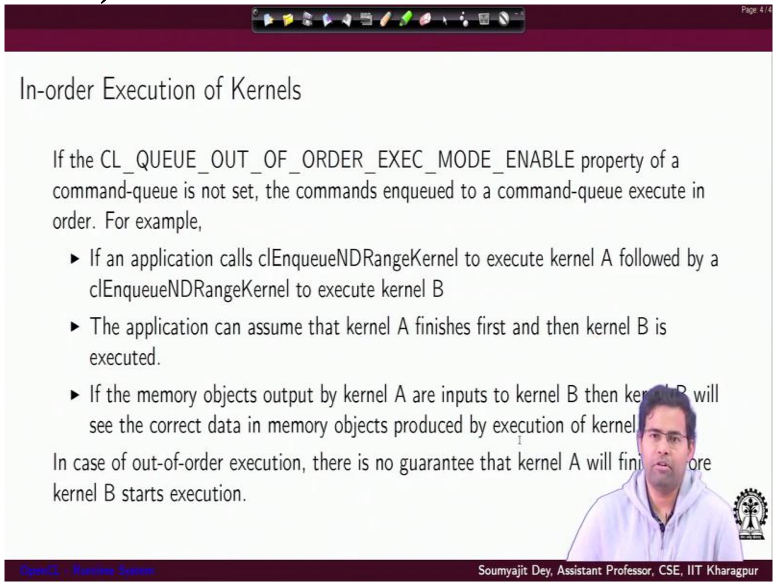
But, if I want, I can give the command queue and out of order property. In that case, the runtime system will decide whom to launch when. And you may be wondering why somebody would want to do that, because the runtime system we often have certain units which are free, but the command which is sitting in front of the command pipeline may not be for those units, those compute units but for somebody else.

Whereas there may be succeeding commands there who could have engaged the corresponding execution units. For example, there may be separate read write units and compute units the compute unit is not free, but, at the same time I could have included a read of here I could have you I mean taking out a read operation from the queue. And in that way I could have achieved some parallelism in using the runtime system, but for that the command queue should also be able to issue the commands in a out of order way.

And just trying to motivate how out of order execution from the command queue may give me better performance. So, the properties are whom as in these cl create command queue with properties has to be set to an in order an out of order or an out of order flag. And so, in order to specify what should be the execution order of commands from the command queue in out of order execution mode, there is no guarantee that the include commands will finish execution in the order they are queued as we have been discussing.

And so the takeaway from this would be that this is primarily being done by setting the CL create command queue with properties command while setting up the command queue itself.

(Refer Slide Time: 04:10)



In-order Execution of Kernels

If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is not set, the commands enqueued to a command-queue execute in order. For example,

- If an application calls clEnqueueNDRangeKernel to execute kernel A followed by a clEnqueueNDRangeKernel to execute kernel B
- The application can assume that kernel A finishes first and then kernel B is executed.
- If the memory objects output by kernel A are inputs to kernel B then kernel B will see the correct data in memory objects produced by execution of kernel A.

In case of out-of-order execution, there is no guarantee that kernel A will finish before kernel B starts execution.

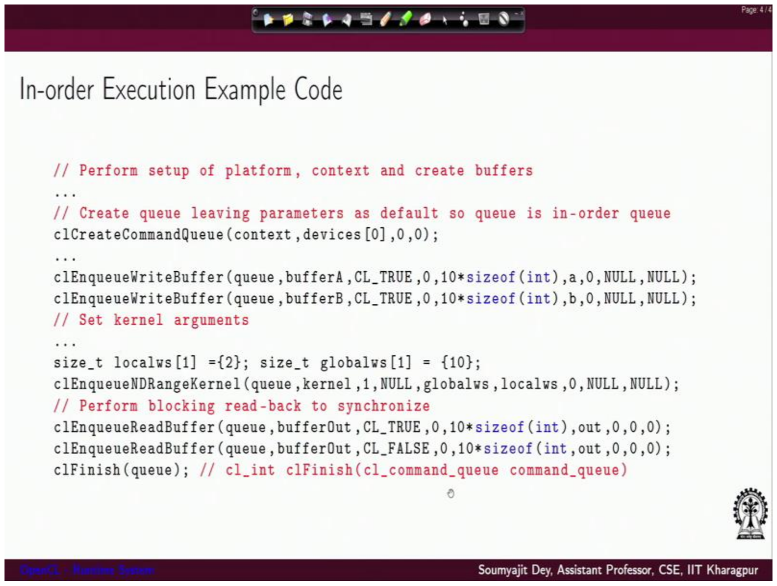
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So we will see if we are talking about in order execution of kernels. Then the cl queue out of order except mode enable property of any command to is not set and the commands in queue are executed in order consider an example these are normal in order execution of kernels were talking about first you can application calls clEnqueueNDrangekernel to execute some kernel A followed by another call to clEnqueueNDrange to execute a kernel B so you have include 2 kernel on launch A and B.

The application may I assume that kernel A finishes first and then Kernel B is executed this assumption from the application side. If the memory objects to be output by A are inputs to kernel B then B will see correct data in memory object produced by execution of it, because in order execution, it will finish and then we will start executing. But in case it is an out of order execution, you do not have such a guarantee that they will finish before be starts execution.

So that is the problem, right. If it is out of order as we said if I make it out of order, then I lose such guarantee. But then again, I will like to sometimes make things out of order, so that the runtime system can give me better performance if possible.

(Refer Slide Time: 05:29)



In-order Execution Example Code

```
// Perform setup of platform, context and create buffers
...
// Create queue leaving parameters as default so queue is in-order queue
clCreateCommandQueue(context,devices[0],0,0);
...
clEnqueueWriteBuffer(queue,bufferA,CL_TRUE,0,10*sizeof(int),a,0,NULL,NULL);
clEnqueueWriteBuffer(queue,bufferB,CL_TRUE,0,10*sizeof(int),b,0,NULL,NULL);
// Set kernel arguments
...
size_t localws[1] ={2}; size_t globalws[1] = {10};
clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalws,localws,0,NULL,NULL);
// Perform blocking read-back to synchronize
clEnqueueReadBuffer(queue,bufferOut,CL_TRUE,0,10*sizeof(int),out,0,0,0);
clEnqueueReadBuffer(queue,bufferOut,CL_FALSE,0,10*sizeof(int,out,0,0,0);
clFinish(queue); // cl_int clFinish(cl_command_queue command_queue)
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, here we have a sample code of in order execution. So, there are certain parts which are commanded because we are just training what are the functionalities you have to do there and those functionalities are things that we have already discussed earlier. So consider that you have already set up the platform context and created the required region write buffers and you have created the queue living parameters as default so that the queue is an in order queue. So the command queue was already there.

I mean, so this is something we doing here and this command is saying that you have already performed a setup of platform context and buffers and all that. And this is the command through which you are creating the command queue for a device, which has been already discovered and the device ID is stored in these device, the device belongs to this context, which has also been defined and other parameters are being set to default to maintain that desire in order queue.

So, you Enqueue a ride buffer command, you Enqueue another write buffer command. So essentially you will be copying this buffer A and buffer B right you are including commands for that in the command queue. Then you will have suitable commands for setting carnal arguments. Once the kernel arguments are set, you will set the global and local work sizes as usual in the OpenCL host program. And then you NDrange here EnqueueNDrange on one of the kernels so let us say it is kernel the first kernel in the queue, right with respect to global and local work size.

And then you perform a blocking read back to synchronize. So essentially, I am just trying to say that after this command, you have a clEnqueueND a read buffer command for the same queue, where you are going to read back the outputs of the kernel right. And while having that read buffer command, you said the blocking read property to true so that unless this command executes, the host program will not move forward to the next command.

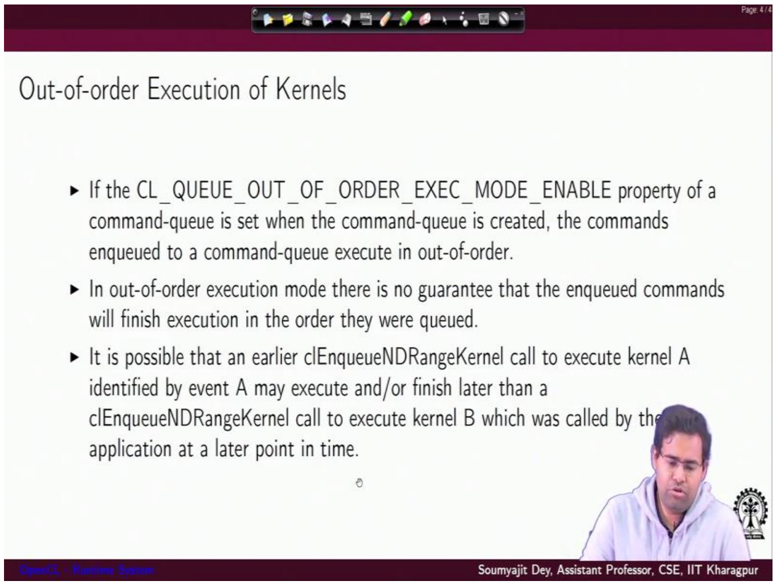
So here you are achieving synchronization using the blocking grid. The blocking grid is ensuring that you do not go beyond to the next kernels read buffer commands. Unless this happens, right. So that you again you may have read buffers performed here clEnqueue read buffer like this and then again a clEnqueue read buffer. So, these are the blocking reads who are performing right once this kernel has executed.

And after that you will see you have the clfinish command. Now, the clfinish command will again ensure that inside this queue whatever commands have been include have all finished so far of course, the blocking read will these blocking reads if they are running routes then anywhere you cannot go beyond each of them. And this additionally says that will I you can go beyond this point only when all these commands are finished.

So, this is the way you will ideally write the program in case you have a simple in order code. So, just to summarize like you have a normal in order execution of kernel, the application can assume that kernel if you finishes first and then kernel B is executing just as we have discussed and the memory objects which are output by kernel A are inputs to be and then we will see the current data and objects produced by the execution of A, I mean, so, if I want a normal in order execution.

So, what will I do is after A has finishes after A has finished, let us say, I am having a blocking read and then if required, I will again to a blocking right so, in that way I can synchronize to the host program or here I am performing a blocking read back to synchronize from the buffer outs.

(Refer Slide Time: 09:43)


Out-of-order Execution of Kernels

- If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is set when the command-queue is created, the commands enqueued to a command-queue execute in out-of-order.
- In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued.
- It is possible that an earlier clEnqueueNDRangeKernel call to execute kernel A identified by event A may execute and/or finish later than a clEnqueueNDRangeKernel call to execute kernel B which was called by the same application at a later point in time.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And then, let us come to the other option, which would be this out of order execution of kernels. So, for out of order execution of kernels, you will have this CL QUEUE OUT OF ORDER EXEC MODE ENABLE property of the command queue is set at the write at the time when you were creating the command queue right. So, in that time, the commands included in the command queue may execute out of order. So, out of order as we have discussed earlier that you do not have any guarantee that the few commands will finish execution in the order they are Enqueue, right.

I mean, so, they will start executing outside out of the order it is possible that an earlier NDrange kernel call to execute kernel A identified by some event A may execute and are finished. So later then a clEnqueueNDrangekernel to execute kernel some kernel B which is called by an application at a later point of time. So it may happen that you I mean whatever commands you have execute include there.

They are executing outside that are normal order, enqueued to a command queue execute. So maybe you have enqueued you have executed this two commands clEnqueueNDrangekernel for a kernel A and against clenqueueNDrangekernel for a kernel B but the kernel B gets actually launched to the device before kernel A has been launched or kernel has been launched, but it is still not finished just to go back why this issue was not there in order execution for primarily for this blocking read.

So, suppose you have a code where you have copied you have written 2 buffers again to the device memory and you have executed a kernel which would have worked on the buffer A and B and then you are reading the buffers performing or blocking read back to synchronize. So, if you have done it in such a way so, this is a blocking read as you can see, and then so that is actually ensuring that the kernel A finishes.

And after that, if you launch another kernel, it is right. But so if you do not if you have an out of order property enable, then you have to actually keep in mind that will the kernels may get launched in a different order, although you have include them in order, but they can technically get lost because again, I will just try to remember remind you this program is what it is doing is just saying.

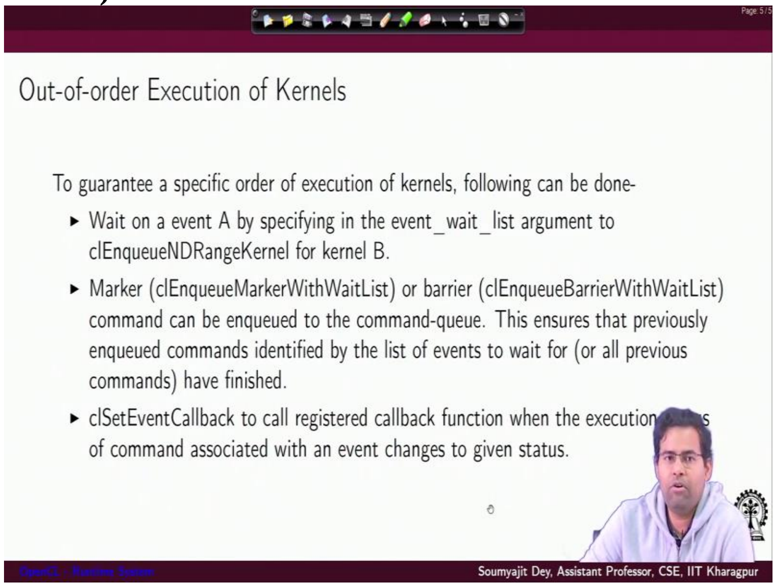
What is the order in which I am including the kernel here 1 kernel is in queue, maybe at some point I am going to enter into another command. The host program is executing commands in this sequence right. So this is just a sequence in which the host program is executing inside the command queue, all these right buffers kernel again, the read these are all common that are getting Enqueue in the command queue.

So this is the execution order of the host program, right is execution order of the host program. But when I have in order execution from the command queue, then these commands are submitted to the device exactly in this order. Only in that case, I have execution in order. Otherwise, if I have an out of order setting in the command queue, then just by in queueing them in order to the host program I do not have a guarantee.

Because still in that case, you see, maybe I am performing a blocking read, but does not matter. This indeed inch content has been Enqueued here, right? It does not mean it is being issued to the device or it has started execution it will finish and then it will start right here by performing the blocking read or by the CL finish all that we are doing is that we are delaying the incoming of the second kernel in the command queue right because we have a guarantee that we have the execution will be in this order right.

But if the command queue is set in an out of order and if I am not having this sequence of commands to be executed, I may not have such a guarantee. So, we will see that how what happens in that case.

(Refer Slide Time: 14:29)



Out-of-order Execution of Kernels

To guarantee a specific order of execution of kernels, following can be done-

- Wait on a event A by specifying in the event_wait_list argument to clEnqueueNDRangeKernel for kernel B.
- Marker (clEnqueueMarkerWithWaitList) or barrier (clEnqueueBarrierWithWaitList) command can be enqueued to the command-queue. This ensures that previously enqueued commands identified by the list of events to wait for (or all previous commands) have finished.
- clSetEventCallback to call registered callback function when the execution status of command associated with an event changes to given status.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, to guarantee a specific order of execution of kernels, this falling can be done. So, you may wait on an event A by specifying the event in the event list argument to the cleEnqueueNDrangekernel for kernel B. So, for every kernel, I can have an event where at least that only when those events as you must specific value, then the kernel will execute. So, I can have all the other options would be that I put in a marker or a barrier using the clEnqueuemarkerwithwaitlist or clEnqueueBarrierwithwaitlist.

So, this as we have seen that I can enqueue barrier on marker commands in the command queue. If I put in such markers then they will answer any ensure synchronization inside the command queue and prevent some out of order executions scenarios which I would want to avoid those which I want to happen to enhance performance because I can keep those that I want to avoid that I can enforce a suitably positioning the markers are the barriers in a command to us.

This ensures that previous to include commands identified by the list of events to wait for have finished or all previous commands I would say. Now the other event, either way will be as we have discussed that I can have a callback function registered and I can have CL set even callback so that when the execution status of a command associated with an event changes to a given status, then this function would actually let the system know and I can make some progress forward right.

So, whatever synchronization primitives we have discussed earlier, they may also be used to actually provide sequencing inside command queue, in case the command queue is already set for out of order execution. So, again I will just repeat if you have in order execution property for the command queue that is fine. So, commands will be getting executed in the sequence. And so, I do not then I do not have a problem, because you have writes followed by 1 corner executing then you can have a blocking read.

So, unless that read is finished, your host program does not go to the next including operation right. So, and there is also important we will need to say that suppose I was not having a blocking retail then what happens? Well, the host program is following this sequence. It will include the read buffer command, then it will also include the letter say you are also including the next corner for executing.

So again, the host program is just Enqueuing right? So it is not waiting for the read command to complete before it will possibly include the next kernel after at any of these points right. Now, due to that, even before all the reads happening to an input buffer for the next Kernel, the kernel may get launched that is why to ever that you will like to have the read set to a blocking read property fine.

So, in that way, in this code, as you can see, we will have this blocking read and then the blocking read would be followed by another ND range kernel command for the next kernel right. So, this is about the in order execution. And then just to summarize, in out of order execution, you have a lot of problems, because unlike the in order execution, where the commands you are Enqueueing in the order in which the host program is executing.

And the actual execution and actual launch of commands are going to follow that exact sequence that gets broken, right because in the out of order execution, although your host program will move like this, and enqueue commands in a specific order, the commands may actually get start executing in a different order because it is an out of order execution. So, this is something again, I will keep on repeating I am in enqueueing commands in order here.

Since I am enqueueing commands in the order here, and the command queue is set to in order execution, the commands will also be launched in order. But here also you have to be very careful. Again, I am repeating I know, but is important. I have to be careful because I am launching the commands. I am including them in order. So that will ensure the commands get launched in that order.

But still, it does not ensure that the previous command finishes before the second command executes because they are getting launched in the order right. For example, as I said, this read followed by the next kernel launch, there is a reason why I make the reader blocking read so that the data is available for the kernel launched even in case of in order execution. But as you can understand the problem is far more pronounced in case of out of order execution.

Because the host problem is in feeling in an order. But even though you are enqueue it an order your commands may not be submitted for execution in that specific order. And so, even in order execution we had that issue right that you may yes, you may be submitting them for execution in a specific order, but you are not waiting for previously issued commands to finish. Here the issue is even more I mean grave I would say, because first of all, you are not even submitting commands for execution in the host specified order.

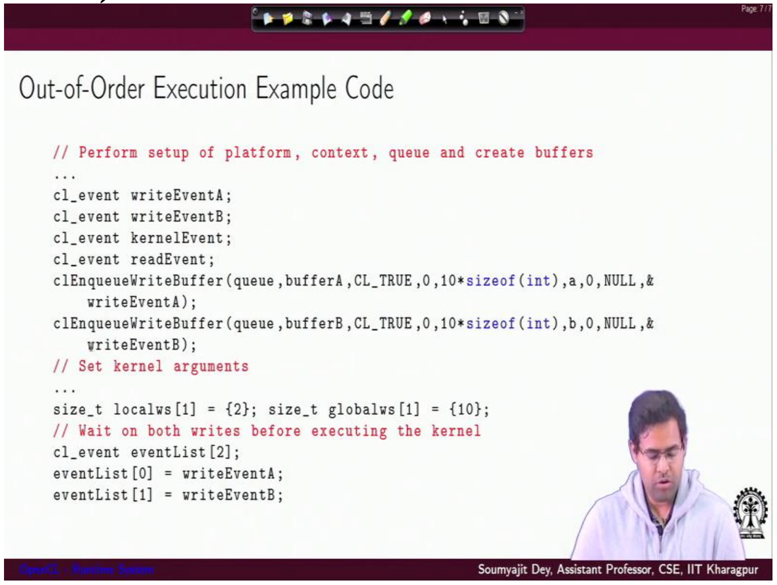
And again, so, you are submitting them in some order to the runtime system wants, apart from the order in which the hostess include. Moreover, the order in which they start execution may not be same as the order in which the finished execution. So, that would mean this kind of a bad scenario in a worst case I would say, let us say the host has include commands in an order. We in order I had already have a problem that without let us say this is first execution is second issue is first issue.

But, the second if the second issued command has a dependency on the first command, for example, or read event has to be completed before a kernel has to execute, then I have to either put in a blocking call or I have to put in a barrier or I have to put in some other synchronization prior to all that. So, that is already the problem in order. Now, you have the additional problem that actually, although you may have the first command, followed by the second command.

When they are getting issued, maybe the second command has been issued, followed by the first commendation has been issued. And that is actually followed by the third command that is issued. And maybe the second command has a dependency on the first command. So the second has been issued, first has been issued third has been issued. This time and the second have a dependency on the fast that is also not specified.

So, you may have multiple issues right because the second having a dependency on first was already the problem and in order for that you have to choose to use blocking and all that here, you are Further complicating the problem with this out of order execution, because you have to somehow make the system know that 2 has to depend on 1 and all that. So, how to ensure this as we have discussed that we will be using we will be making use of event wait lists, we will be making use of markers and barriers, and we will be making use of callback functions.

(Refer Slide Time: 22:46)



Out-of-Order Execution Example Code

```
// Perform setup of platform, context, queue and create buffers
...
cl_event writeEventA;
cl_event writeEventB;
cl_event kernelEvent;
cl_event readEvent;
clEnqueueWriteBuffer(queue,bufferA,CL_TRUE,0,10*sizeof(int),a,0,NULL,&
    writeEventA);
clEnqueueWriteBuffer(queue,bufferB,CL_TRUE,0,10*sizeof(int),b,0,NULL,&
    writeEventB);
// Set kernel arguments
...
size_t localws[1] = {2}; size_t globalws[1] = {10};
// Wait on both writes before executing the kernel
cl_event eventList[2];
eventList[0] = writeEventA;
eventList[1] = writeEventB;
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

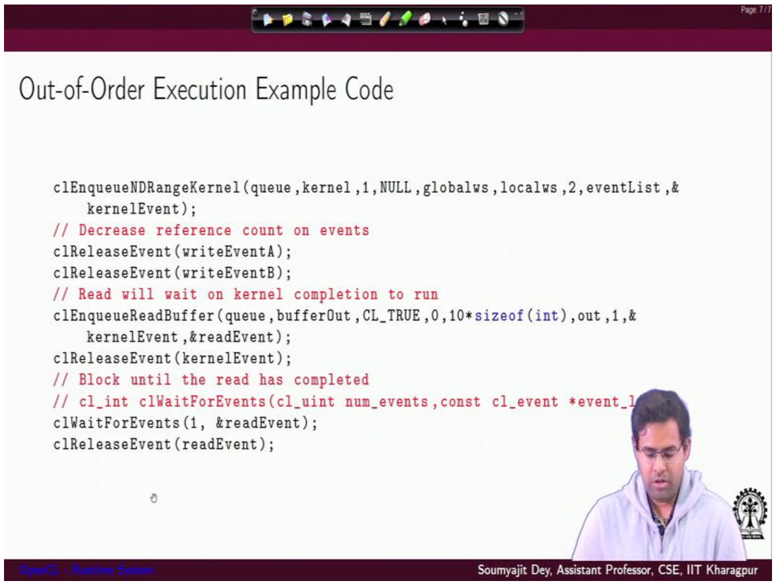
So let us see what are this? So pencil will first of all, let us consider that we have already set up the platform contest Q and the buffer circulated as usual practice and you have some open CL events defined as write event A write event B and you have an event defined for kernel event, you have an event for event and all that. Now, we will start creating this normal enqueue commands and we will be specifying some events to be attached to them, which we have we are not doing earlier.

So let us say we have a right buffer command. So, the So, the right buffer command in the in a defined command queue, you are writing the buffer you are including a command to write the buffer day into a device I mean in for the specific device to is the command gives attached blocking rights, you have the size and other specifications and you attach a command away an event right event A with this.

So that the status of this command can be found from the status of this write event A open set event similarly, you also attach the right event B for the other write command other I buffer command for buffer B. So, you have include 2 write commands in the buffer in the command queue they are writing in the device memory buffer A buffer B the status of this rights can be known from the value that the runtime system will as you actually assigned to write event A and right even B events.

After this you have usual definition of kernel arguments your usual definition of local and global work sizes. And now, let us say you create an event at least by packing these 2 events write event A and write event B into this era event least.

(Refer Slide Time: 24:36)



Out-of-Order Execution Example Code

```
clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalws,localws,2,eventList,&
    kernelEvent);
// Decrease reference count on events
clReleaseEvent(writeEventA);
clReleaseEvent(writeEventB);
// Read will wait on kernel completion to run
clEnqueueReadBuffer(queue,bufferOut,CL_TRUE,0,10*sizeof(int),out,1,&
    kernelEvent,&readEvent);
clReleaseEvent(kernelEvent);
// Block until the read has completed
// cl_int clWaitForEvents(cl_uint num_events,const cl_event *event_l
clWaitForEvents(1, &readEvent);
clReleaseEvent(readEvent);
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, why would we want to do that you want the kernel enqueuing to happen in depending on whether this to write events have been have actually completed. So, what you do is now you enqueue the kernel for execution in the command queue providers parameter settings global and local work sizes and you also give it a eventless to wait on and you attach with this command the kernel event.

So if I try to draw a graphical picture here, what is happening this is the queue so you have executed you have enqueued a write buff command. You have another write buff command. And you have enqueuer the kernel with this write buffer command. So these are 2. These are the 2 right buffer commands and talking about this I has been enqueue right now with this 2. We have attached to events, OpenCL events. And those 2 events live events. Now start appearing into the event list of this kernel. What are those OpenCL events write event A and right event B.

So, these are OpenCL elements write event A and similarly B and they are here in the event list A, B. So, now you do not have the problem right because the event leads to something like only when these events are finished they reach their CL finish only then this command which has been enqueued can execute. So, now, even if the command queue start I mean the runtime system try to optimize execution of the command queue. Even if it tries to initiate dispatch of this kernel before this it cannot, because now you have explicitly specified the dependency of this write event A and write event B.

This should be write event A and this is B here explicitly specified the dependency of write event A and write event B with the execution of the kernel using this signals are the OpenCL events write event A and write event B. So, now, when these 2 are done there they actually force that even this cannot execute out of order, it can only execute after i read even then and write event B done and they reach the status of clfinish.

Now, when they are done, you see this ND range kernel command when you are enqueuing this in the queue. In the last argument, you have the kernel event. So that is another the last flag that would mean that with this command itself, you are attaching another event called kernel event. So that means with write event A and write event B finishing, they can start in new order right I have not given any order it can is an out of order queue that these 2 can start in any order.

When they finish only then the kernel execution can start which is being forced by this event list here, when the kernel executions ends, then you have also associated this kernel event type OpenCL event kernel event with the queuing of this ND range kernel the queuing of this kernel. So, now, only when the kernel execution finishes, then this kernel event will get this finished status and this can be used for upstream further execution.

Which are dependent on the kernels execution. So, for example, after this point, you do not have any importance of write event A and write event B so you can just release those events right. So you do a CL release event. That is an OpenCL function. Now you want the second kernel to execute right. So, when do you are so you want then you, want to read the output that has been computed by this kernel.

So, for that you have in you will enqueue a read operation. So, now you will in queue read operation right. So, you have a read operation. Now, this read operation is waiting for this kernel event and this read operation will output a rudiment so, that is the associated so, maybe you will say that it is waiting for the kernel event and it is associated with this read event right. So, when this read operation is included.

It is restricted from starting earlier by this dependents on the kernel event right this is a dependency on the kernel event only when this is done then read operation can start when read operation means there associated even read event will be true. At this point you do not have any requirement of the kernel event so you can release that event. And you can actually use this to defend to have some further dependent executions going on here, right.

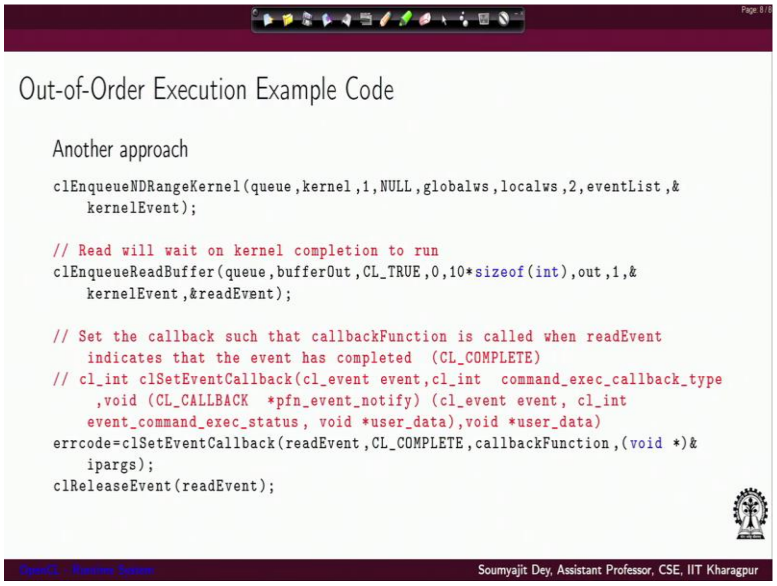
If you want them you can any few more commands with the dependency of this. So, here if you want to block until the read has completed, your option would be you use the function CL work for events, unless you are you do not want to give another command which will depend on the redefined you just wait for this event using the CL wait for events right. So you just wait here. So the option we choose here is we are not enqueuing any more command so you are just waiting.

And then when it is done, then this function is a blocking the execution here, it will let the execution progress and then you release everything. So here we are showing that how using these events and dependency lists of every enqueue ND range command. So this enqueue commands earlier, we have been just enqueuing commands into the queue. But now you are showing how enqueue can be done with dependency lists.

The last 2 arguments of every enqueue command used for this event dependency list. The last argument is for the output dependency that well this is done this event will now go on. So this is actually specifying the event that you are attaching with the command that we have been enqueued. And these events status will keep on changing with the different execution status of this command.

And before this event, the last but one or movement is the event list. This event is may contain a set of events and when all those events actually get the clfinish. Then only this only command can start executing so this is how you can mark out dependencies and force an ordering.

(Refer Slide Time: 32:01)



Out-of-Order Execution Example Code

Another approach

```
clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalws,localws,2,eventList,&
    kernelEvent);

// Read will wait on kernel completion to run
clEnqueueReadBuffer(queue,bufferOut,CL_TRUE,0,10*sizeof(int),out,1,&
    kernelEvent,&readEvent);

// Set the callback such that callbackFunction is called when readEvent
    indicates that the event has completed  (CL_COMPLETE)
// cl_int clSetEventCallback(cl_event event,cl_int  command_exec_callback_type
     ,void (CL_CALLBACK  *pfn_event_notify) (cl_event event, cl_int
    event_command_exec_status, void *user_data),void *user_data)
errcode=clSetEventCallback(readEvent,CL_COMPLETE,callbackFunction,(void *)&
    ipargs);
clReleaseEvent(readEvent);
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

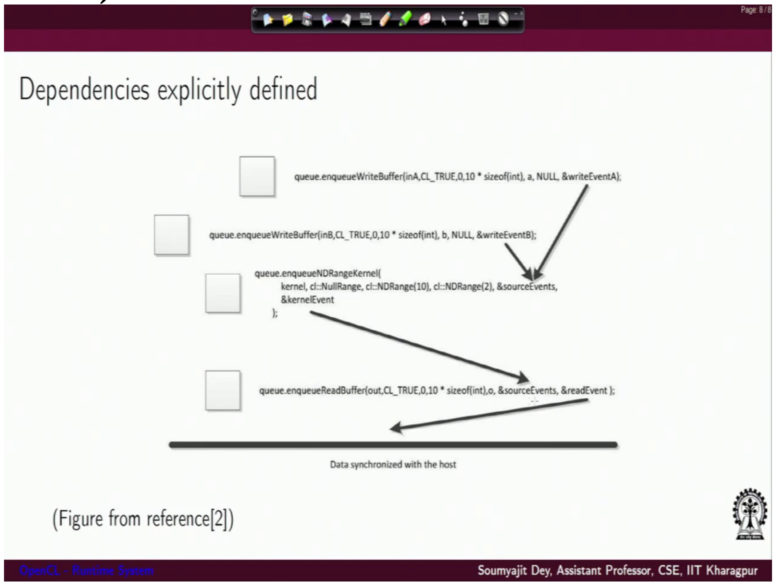
Now, there could have been another approach, which is that you suppose you have this clenqueueNDrangekernel. So you enqueue the kernel, you have the event list, and you have the kernel event. So we are still speaking from that point. Now, the read will wait on the kernel competition to run. After that you use this kernel event to include the 2 as a dependency to include the read event. When the read event is done, you get the redefined call. So the read event status will turn CL complete right. Here also you had the read even up to this is fine. After that you were doing a wait for events instead of wait for events.

Alternatively, what you could have done is that you could have used the callback function. So this is our approach we thought, right? So what the callback function will do is it is indicating. So that callback function will be registered with this event using this CL set event callback primitive. So all you do is with this event CL read event with this event read event you are associating a function called callback function.

So the way to read it will be when this read event will get the status CL complete, then this CL function this callback function will execute asynchronously, with some arguments. So, the definition of the function prototype clSetEventCallback is that it is the event to whom you can look that whether this event has attained these execution status. When it has when some event like in this case, the reed event has attained an execution status, in this case CL complete.

Then some callback function like this of type CL callback. In this case, we are just calling the callback function name as callback function will execute with some argument whose which can be specified here right now, so in that way I can have another callback function to execute. And finally, I will just release the read event that we have defined earlier right. So, that can be another way to synchronize things in out of order execution.

(Refer Slide Time: 34:23)



Dependencies explicitly defined

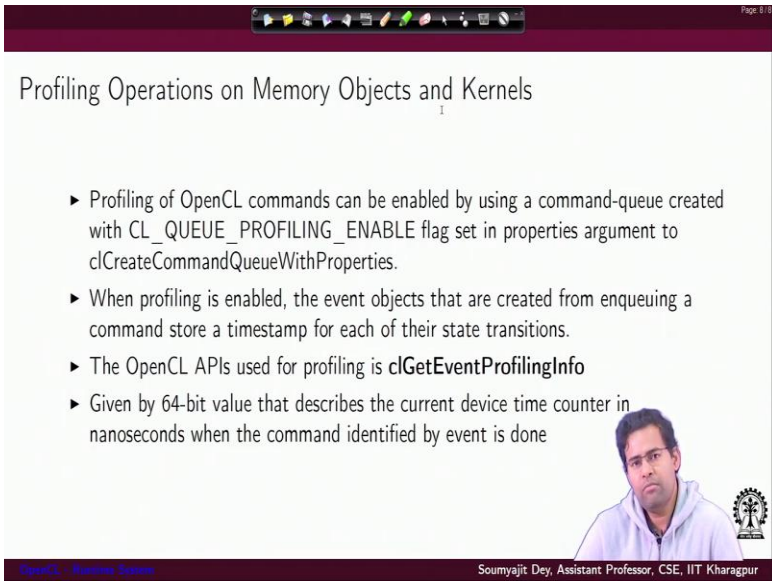
```
queue.enqueueWriteBuffer(inA,CL_TRUE,0,10 * sizeof(int), a, NULL, &writeEventA);

queue.enqueueWriteBuffer(inB,CL_TRUE,0,10 * sizeof(int), b, NULL, &writeEventB);

queue.enqueueNDRangeKernel(
        kernel, cl::NullRange, cl::NDRange(10), cl::NDRange(2), &sourceEvents,
        &kernelEvent
     );

queue.enqueueReadBuffer(out,CL_TRUE,0,10 * sizeof(int),o, &sourceEvents, &readEvent );
```

(Figure from reference[2])

So, these are pictures we have taken from this reference, which we have let us reference is the open CL is a nice book on OpenCL and OpenCL 2 points where I would say so as you can see that we have this write buffer commands and these are the associated events that we have defined. The ND range kernel is the include kernel here enqueuing this enqueuing ND range kernel is enqueuing this kernel it can only execute when these events are done.

And it will output the kernel event to CL complete and using this kernel event. I can initiate the execution of read buffer and when the buffer is done it will get as output the redefined and in that way of the data synchronized with the host. So, this is the other option of dependency.

(Refer Slide Time: 35:12)



Profiling Operations on Memory Objects and Kernels

- Profiling of OpenCL commands can be enabled by using a command-queue created with CL_QUEUE_PROFILING_ENABLE flag set in properties argument to clCreateCommandQueueWithProperties.
- When profiling is enabled, the event objects that are created from enqueuing a command store a timestamp for each of their state transitions.
- The OpenCL APIs used for profiling is **clGetEventProfilingInfo**
- Given by 64-bit value that describes the current device time counter in nanoseconds when the command identified by event is done

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

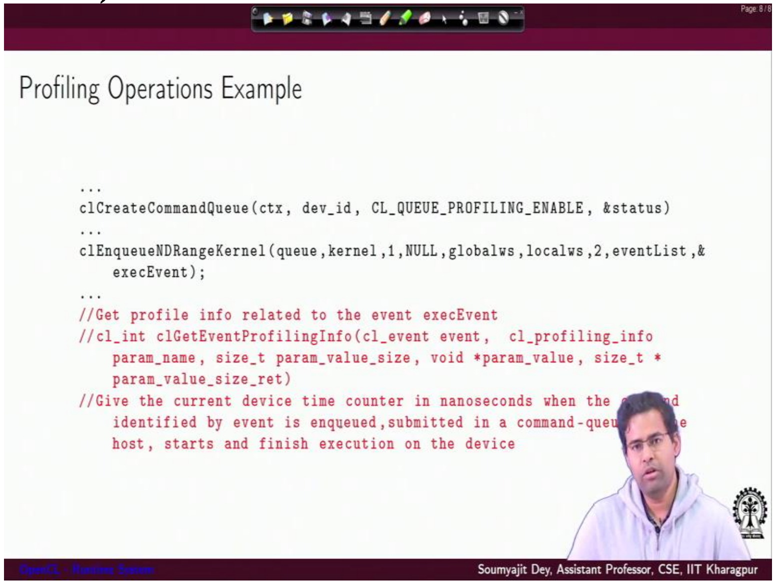
Apart from this, we have an option of profiling operations on memory objects and kernels in OpenCL like so, the way it works is you can profile OpenCL I mean, in case you want to profile the execution of OpenCL program, you have to set a command queue with CL queue profiling enable flag. Now, once that is done, and then the event objects that are created in the queue from enqueuing of a command, they will I mean, they can also store as timestamp for each of their state transitions.

So, just to remember, with every command that you enqueue, you can have an associated event object for that event object. You can have as we know that the runtime will keep on changing the status of the event object right. Now, as we have discussed earlier that every enqueue command will return an event right. Now, if you actually give a name in that field of enqueuing of CL enqueue commands, then for a specific event.

Then that return event will be in the name in that event, right. And so, you can use that event name to sample the different possible status like CL complete CL finish CL submitted CL pending like that to know what is the status of that command. Now with profiling and I will, the system will also I can also give you a timestamp when the status changes from something to something.

So for this you can use the OpenCL API clGetEventProfilingInfo. So give is given by a 64 bit value that describes the current device to encounter in a nanosecond resolution that you can get to know when some event can attain a specific status.

(Refer Slide Time: 37:09)



Profiling Operations Example

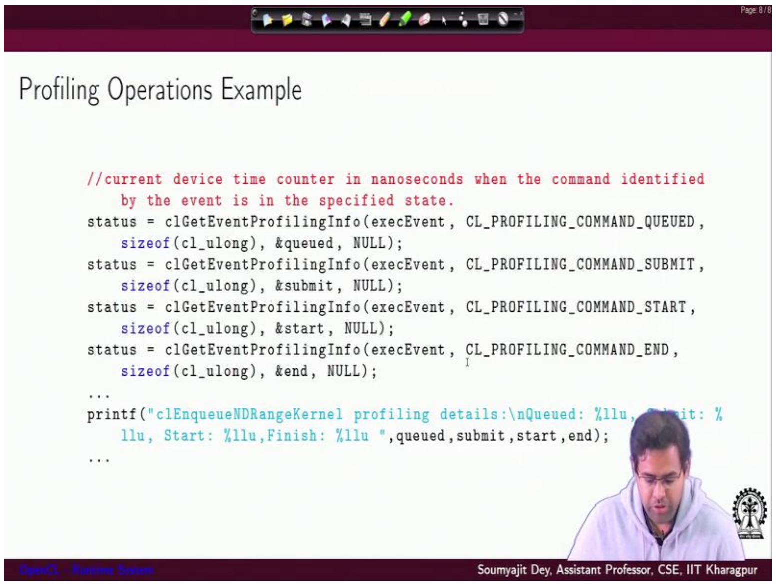
```
...
clCreateCommandQueue(ctx, dev_id, CL_QUEUE_PROFILING_ENABLE, &status)
...
clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalws,localws,2,eventList,&
    execEvent);
...
//Get profile info related to the event execEvent
//cl_int clGetEventProfilingInfo(cl_event event,  cl_profiling_info
    param_name, size_t param_value_size, void *param_value, size_t *
    param_value_size_ret)
//Give the current device time counter in nanoseconds when the c    nd
    identified by event is enqueued,submitted in a command-queu      e
    host, starts and finish execution on the device
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Let us take an example suppose you have created a command queue and it is in profiling enabled mode. Now, you are enqueuing a kernel in that queue and that kernel has event list and you are attaching an execEvent and with that kernel now, you So essentially, this will help you I mean, the I mean, since you have the queue with profiling and even more, so, the runtime system will ensure that for this execEvent, during the lifetime of this event, whenever it is switching these different modes, the corresponding timestamps are collected by the runtime system.

And you can use profiling information of API to collect those values. So you can use this CL get even profiling info function to get those values. And so, what we can do is give the current time count device time comes in nanoseconds so that is what it does, and it will sample it out when the command identify the event is include submitted in command 2 by the host and starts and finishes execution on the device. So, these are the different states through which the command will go through. And although state whenever those states are reached those values can be collected.

(Refer Slide Time: 38:17)



Profiling Operations Example

```
//current device time counter in nanoseconds when the command identified
    by the event is in the specified state.
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_QUEUED,
    sizeof(cl_ulong), &queued, NULL);
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_SUBMIT,
    sizeof(cl_ulong), &submit, NULL);
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_START,
    sizeof(cl_ulong), &start, NULL);
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_END,
    sizeof(cl_ulong), &end, NULL);
...
printf("clEnqueueNDRangeKernel profiling details:\nQueued: %llu, Submit: %
    llu, Start: %llu,Finish: %llu ",queued,submit,start,end);
...
```

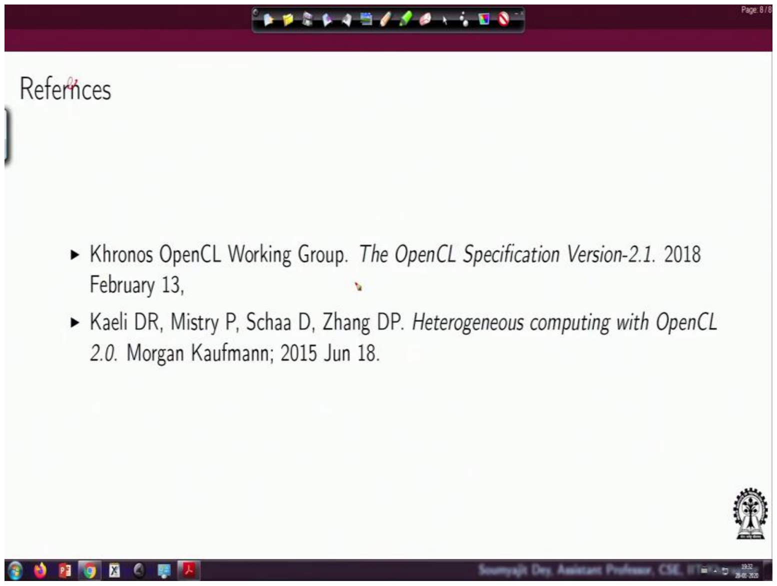
OpenCL - Runtime System Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

For example, you can use this CL get event profiling info API for this exact event even and see that we I mean, so, this is the way you have to do it. So, for this you have to write it likes execEvent and is the event and CL profiling command queue. So, you so essentially, you should return what is the exact time when this exact event reach the status that it has got actually queued? What is the exact time when it has been submitted for execution?

What is the exact time when it starts execution and what is the time when the command is sent? So these are the different statuses that can be example and when the status is an example, they would actually get returned in this. So, this is basically the time so CL profiling command and so, you have queuing time submitting time submission of for execution to the device, start time for execution, actual execution and the intent for execution.

And all these values get stored in this corresponding addresses for queue submit start and end as we have been defining here. So it is this field, right? It is this field parameter value field in which those values get stored. And finally, if you want to print them, you can just use and use this as a normal printer function and print the exact timings for that specific kernel getting queued. The kernel getting submitted the kernel is it starting and the continuity is going to finishing because all of them have got stored in these variables.

(Refer Slide Time: 40:06)



References

- Khronos OpenCL Working Group. *The OpenCL Specification Version-2.1.* 2018 February 13,
- Kaeli DR, Mistry P, Schaa D, Zhang DP. *Heterogeneous computing with OpenCL 2.0.* Morgan Kaufmann; 2015 Jun 18.

So these are some important references we did like to you may want to visit for knowing in more detail about OpenCL. So, we have made exhaustive use of these two resources. The first one is the open CL specification made by Khronos, OpenCL Working Group Khronos is freely available in the internet. The second one is a very nice book on heterogeneous computing with OpenCL by these authors as written here. So I hope this was a nice journey to the basics of OpenCL. Thank you for your attention.