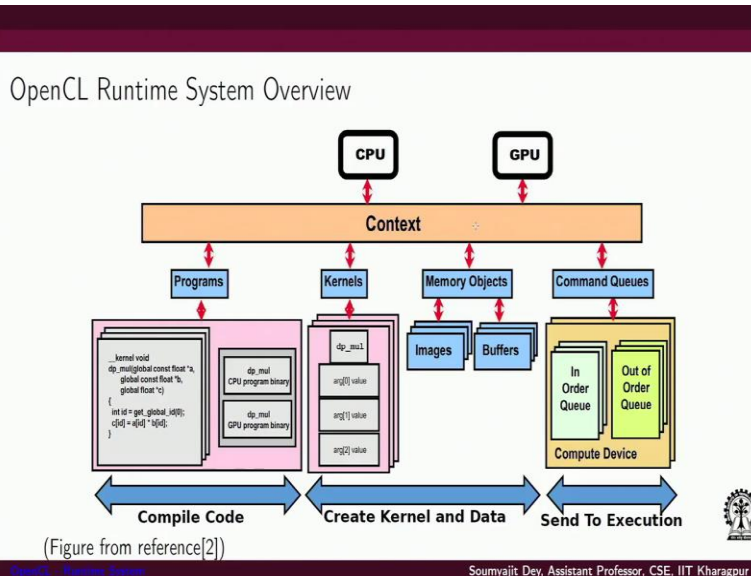


GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 46
OpenCL - Runtime System (Contd.)

(Refer Slide Time: 00:33)



(Refer Slide Time: 00:34)

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;
}

```

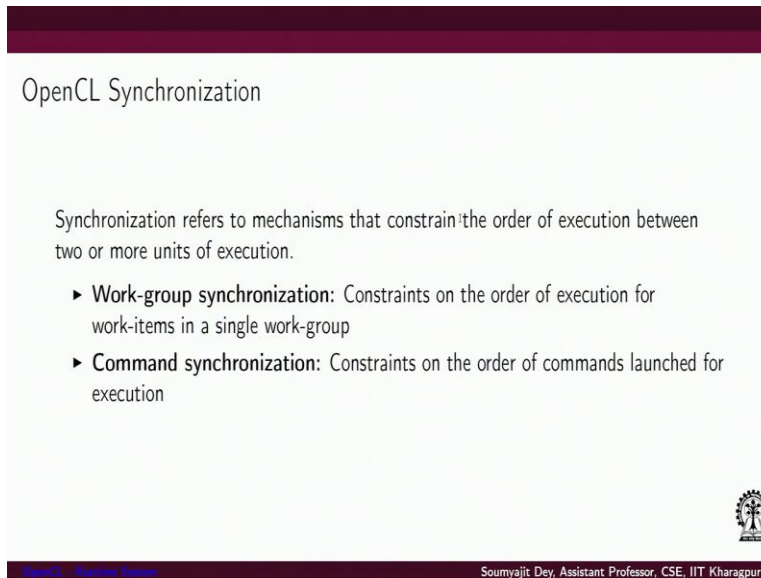
IIT Kharagpur logo

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

Hi, welcome back to the lectures on GPU architectures and programming. So I believe in the last lecture we provided you with the basic open CL runtime overview. And we gave an example of a complete open CL program enqueueing the host code primarily the host code I would say like

how to compile the code how to build a create the kernel, the different data buffers memory objects, and finally, set up a context and create command queues and then execute the kernel by issuing suitable read it read kernel launch and write back and find out final rate commands and all that.


(Refer Slide Time: 01:02)



OpenCL Synchronization

Synchronization refers to mechanisms that constrain the order of execution between two or more units of execution.

- ▶ **Work-group synchronization:** Constraints on the order of execution for work-items in a single work-group
- ▶ **Command synchronization:** Constraints on the order of commands launched for execution

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

So with this, we will now move over to the next topic, which is again, related to synchronization in open CL so if you remember in work coverage of CUDA programs also we had significant coverage on synchronization among threads. And it is a far more involved topic I would say, when we are talking about open CL synchronization. So here, it primarily refers to mechanisms that will constrain the order of execution between 2 or more units of execution like.

So just like if you remember in CUDA the synchronization primitive we are primarily using was seeing thread using which we could actually synchronize among threads inside a thread block. And if we wanted to actually have global level synchronization that has to be managed through multiple launches of kernels through the host program. So equivalently in open CL, we have this concept of synchronization of work groups, which essentially constraints on the order of execution for work items inside a single work group.

So these are concepts, which is essentially equivalent to thread block synchronization in CUDA. And, of cworkse, we also have this idea of synchronization of commands, which gives

constraints on the order of commands launched for execution. Because if you remember in open CL, at the end, what you have is a command queues which are being set up with this sequence of commands to be executed.

So you do not really launch commands directly, but rather, you are just issuing commands enqueueing them into the queue. And it is the runtime systems job to pick up commands from the queue in that order and execute them. So you also have this scope of synchronizing the order of commands launched.

(Refer Slide Time: 02:53)

Work-group Synchronization

- ▶ Synchronization between work-items in a single work-group is done using following command -
 - ▶ `void barrier(cl_mem_fence_flags flags)`
(`work_group_barrier` from OpenCL 2.0 onwards)
- ▶ Options for flags are-
 - ▶ `CLK_LOCAL_MEM_FENCE`
 - ▶ `CLK_GLOBAL_MEM_FENCE`
- ▶ All the work-items in a work-group must execute the barrier before any are allowed to continue execution beyond the barrier
- ▶ Work-group barrier must be encountered by all work-items of a work-group executing the kernel or by none at all

OpenCL - Runtime System

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So let us first move into the idea of group synchronization. So this is essentially about synchronization between work items as we discussed inside a single work group. So, for this the open CL function will be using is barrier, and for the barrier function takes this cl mem fence flags type variable and from open CL 2.0 onwards, we also have this is a command for work group area.

So, this is a similar command, but it provides you the notion that this is about synchronization inside the work group. And for that you have this a flag as well as another field called scope. For now we will just go with the flag. Now, what are the options for flags? First of all, again, I will just repeat the barrier command or the work group barrier command equivalently open CLs

current version, they will be synchronizing work items inside work group. In the flag part, you have 2 options.

One is this clk local mem fence and you also have this clk global mem fence? The issue with this main fences with that what is the area of the memory you want to make visible to the work items for synchronizing? So whether it is the local memory or the global memory upon that it depends what flight you are willing to choose. All work items in a work group must execute the barrier before they are allowed to continue execution beyond the barrier.

And it must be of cworkse, encountered by all work items of work group executing by the kernel or none at all. Now, this is very important. First of all, let us understand that you have to have it the primary semantics of barrier is that every work item inside work group will reach the barrier and once all of them received only then the any work item can go beyond work group. The next important thing is you may be thinking well if I have a press the barrier inside and if else block, then maybe some of the work items will be facing the barrier.

And that was work items will be bypassing the barrier will that is not true the semantics are very at ease that you have to write the code in such a way that either all the work items are supposed to execute the barrier, I mean, they have to encounter the barrier, or everybody will bypass them. Because the barrier by it is the runtime system, by his definition of the barrier could wait for all work items to, to be encountered.

If it is partial, then it will be working continuously, whereas the other work items are not even facing it because of divergence that will create a deadlock. So the support up to the programmer to write the code in such a way to ensure this last property.

(Refer Slide Time: 05:52)

Work-group Synchronization Example
Kernel Code

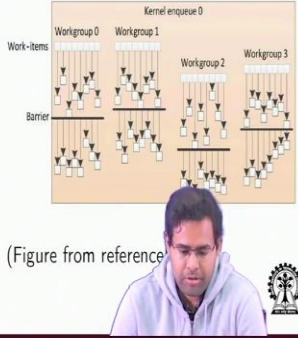
```

__kernel void simpleKernel(__global float *a,
    __global float *b, __local float *localbuf
    ) {
    //cache data to local memory
    localbuf[get_local_id(0)] = a[get_global_id(0)
    ];

    //wait untill all work_items have read the
    data
    work_group_barrier(CLK_LOCAL_MEM_FENCE);

    //perform the operation and save in output
    buffer
    unsigned int addr = (get_local_id(0)+1) %
        get_local_size(0);
    b[get_global_id(0)] = localbuf[get_local_id(0)
        ] + localbuf[addr];
}

```



(Figure from reference)

OpenCL - Beginner's Guide
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, just to recall here about the flags, so I believe the barriers, semantics is clear to you use the synchronization primitive just like CUDA, but also like in CUDA, the synchronization primitive the way you use it, you have to make sure that either all the problems have threads in the work group encounter it or none at all. So that you can avoid the deadlock issue. Now, regarding the fence and the flag, as I said that if you said the flag is local main fence, then you make all the updates in the local memory visible to the fence in the work group.

And similarly, if you set it as global mem fence, then you are essentially sitting in making all the updates in the global memory visible to the work group. So that actually provides you an additional flags, you mean depends on what you are sitting here you are, directing them system that what you really want, what kind of behavior do you want with respect to the memory so the barrier is controlling the way threads synchronize and the memory fence and we are sitting controlling the memory is being feasible at what level to the synchronizing threads.

Also, I would like to add that in, in the newer version of work group barrier, there is an additional flag here called scope. If you are interested you mean if you want to go more into the more advanced concepts of open CL, well, you can go there. I mean, get into that, that and figure out what is this idea of scope that comes with the flags for mem fence. So now we proceed to some example on one group synchronization.

So, let us take this example of a simple kernel. So, first what you do is, you execute the simple get global ID and get local ID calls. Using which you are caching data from the global memory to the local memory. So I hope by now you are familiar with this, get local ID and get global ID calls get local ID calls in open CL. Essentially, they are helping you to figure out what is the global ordering of the threads and what is the local ordering of threads inside the work group.

Using the local ordering of threads you are figuring out in which position of a local buffer, you would be writing a value. So once you have cash the data you want all the all the threads to reach these barrier to me to wait at this barrier until all the threads have done with the previous job assigned to them. And after this barrier synchronization with respect to the local memory updates here, you are going to the next part of the code where you perform some operation and save the output in the buffer.

So here you have defined some operation that you calculate an address based on this get local ID person, get local ID, and then you do something over that value so there is some functionality here, but we are just trying to show that how things are going on. So, if you look into the figure, for this kernel, you have launched multiple while groups, we are trying to convey this fact that this position of the so this is the timescale I would say, in this y axis.

And you have all the while groups being dispatched. As you can see that the barrier is being the synchronization is actually happening at different points of time, across all groups. But inside every work group, all the threads have to hit the barriers synchronize and then move forward.

(Refer Slide Time: 09:58)

Event Object

- ▶ An event object can be used to track the execution status of a command.
- ▶ The OpenCL API calls that enqueue commands to command-queue(s), create new event object that is returned in the event argument.
- ▶ In case of an error enqueueing the command in the command-queue the event argument does not return an event object.
- ▶ Can query the value of an event from the host. For example to track the progress of a command.

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

So, this is one synchronization primitive using barriers that you can do. Now, there are several other primitives we will see, one of the important things is open CL events. So that brings us to this notion of event objects. So an event object is used to track the execution status of an open CL command like as you know, that every open CL every execution is like enqueueing a command in a command queue.

Every commands execution will have the corresponding status and the status is can be pulled by an event. So Open CL API calls and enqueue commands to open CL command queue and creates new event objects, which are returned in the event argument. In case of an error, if there is an error in in keeping the command in the command queue, then the event argument will not return an event object but otherwise it should be successful in returning an event object.


And the API can query the value of an event from the host. For example, you can track the progress of a command by inquiring to runtime system about the status of the corresponding event the value of the corresponding event.

(Refer Slide Time: 11:11)

Event Object

Execution status of an enqueued command at any given point in time can be one of the following:

- ▶ CL_QUEUED
- ▶ CL_SUBMITTED
- ▶ CL_RUNNING
- ▶ CL_COMPLETE



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


We will see some examples. For example, for every in enqueue command, they are these are the following valid status that that Open CL runtime system will provide for example, it can be a it can have a CL queued. So, just to summarize, every open CL command will pass through these 4 statuses. And for this corresponding to the status, you can define an event so that means for every event, it can be attached to a command and the event will have disposable it whether a command is skewed whether it is submitted, whether it is running or whether it is complete.

(Refer Slide Time: 11:51)

Event Object

OpenCL APIs related to Event Object-

- ▶ **clCreateUserEvent:** Create a user event object
- ▶ **clSetUserEventStatus:** Set the execution status of a user event object
- ▶ **clWaitForEvents:** Waits on the host thread for commands identified by event objects in event_list to complete
- ▶ **clGetEventInfo:** Return information about an event object
- ▶ **clSetEventCallback:** Register a user callback function for a specific command execution status. The registered callback function will be called when the execution status of command associated with event changes to an execution status equal to or past the status specified by command_exec_status.



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

Now the Open CL API relates to the event object and there are a few ways in which you can actually do the following functions using which you can use in an events. For example, CL create user event is used to create a user level event object. The status of an event can be set by

CL set user event status, it will set the execution status of work user event object. CL wait for event will wait on the host thread in the host side thread for commands identified by the event object in the event list to complete.

So I mean, there are there may be events for which you are waiting and you want to implement that kind of behavior that can be done by CL wait for events, CL get event info, so this is like a query function which will return information about some event object. Now CL said even callback is related to registering of callback functions for specific command execution status. So there is usage of callback functions is they can exist as synchronously.

So, suppose you want that for a specific command, you want to figure out whether it has reached a corresponding status or not some specific status, whether it is completed whether it is pending or not. And you want that whenever that event happens that it has reached that specific status, then some thread, some specific callback function will be asynchronous, we launched in the host site as a separate thread.

Now, this registered callback function will be called when the execution status of the command associated with the event changes to some specific execution equal to or pause the status that is specified by command exists at us. So just to make it very simple, using this function, CL set event callback, you can command a system that worked. I want some specific function called a callback function to be launched when the execution status this command becomes some things that I specify.

So when that equality is holding, that means the execution status of that command is equal to whatever I have specified, then this specific callback function will start working. Now, why is that normal function not a normal function, but I call it a callback function because it will execute because the host does not wait for it, it will be called and it will start executing asynchronously in the host side will understand it at the end with some examples.

(Refer Slide Time: 14:31)

The image shows a presentation slide titled "Command Synchronization". The slide contains a bulleted list of points. At the bottom right of the slide, there is a small video inset showing a man in a white hoodie. The slide is part of a presentation, as indicated by the navigation icons at the top and the footer text.

Page 1/1

Command Synchronization

- ▶ Command synchronization is defined in terms of distinct synchronization points.
- ▶ A synchronization point between a pair of commands (A and B) assures that results of command A happens-before command B is launched
- ▶ The synchronization points occur between commands in host command-queues and between commands in device-side command-queues.
- ▶ All OpenCL API functions that enqueue commands return an event that identifies the status of command.
- ▶ Value of the event associated with the command is set to CL_COMPLETE when done

OpenCL - Runtime Overview

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So here I am just trying to introduce the definitions and concepts. Now, the next important thing that why do we use this idea of events we will figure out that they also help us to provide synchronization of commands. Now, they can be performed by in terms of distinct synchronization points. A synchronization point between a pair of commands ensures that results command happened before command this launched you may always want that I have include commands in multiple command queues.

But the execution of some command in someone queue should depend or should have should only start when something else in some other commands you may have happened, you may want some such dependencies to be satisfied. How do you do that? Well, you have to synchronize between commands. So let us try and understand this is different from synchronization among threads, like we have done in CUDA, and like we have seen as examples in open CL.

But here it is a bit different concept. We are trying to synchronize among commands that have been enqueued in the command queue. The synchronization points occur between commands in host command, queries, command queues, and between commands in device said commanders so synchronization points will occur again, I will repeat between commands in host command queues and between commands in device that command queues will see some examples between them.

And all open CL functions that enqueue commands, return an event that identifies the status of the command. Like we have this earlier that every command, I mean, every command, which is being enqueued for it, an event is returned. And by pulling that event or by querying that event, I can always figure out the status of execution of that command, the value of this event, which is being associated with the command, while enqueueing, the command will be set to CL_COMPLETE when the execution is done.

Now, this is very important like so, I will just summarize that all this idea of events which we should have been defining the primary reason is you are enqueueing a command at that moment you are it will return the event that events will be one of these based on the execution status of the command. And when it is everything finishes, you will it will reach this CL_COMPLETE status.

(Refer Slide Time: 16:58)

Command Synchronization

The synchronization points defined in OpenCL include:

- ▶ **Completion of a command:** A kernel-instance is complete after all of the work-groups in the kernel and all of its child kernels have completed. This is signaled to the host, parent kernel or other kernels within command queues by setting the value of the event associated with a kernel to CL_COMPLETE.
- ▶ **clWaitForEvents:** This function waits on the host thread for commands identified by event objects in event_list to CL_COMPLETE. The events specified in event_list act as synchronization points.

OpenCL - Basics, Syntax

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now this synchronization points that we have defined in open CL they include the following for example, completion of a command a kernel instance is complete after all the work groups in the kernel and all of its child kernels have completed with all work groups must complete. So, or and we need to understand that inside a kernel you will be launching child kernels those also has to be completed. Now, only when that is done this is signal to the host, parent kernel or other kernels within command queues, I mean, so, for every kernel there may be a parent kernel or other kernels.

So, this is signal by the corresponding event that was defined while in enqueueing this kernel execution command. And so, this finishing of the task of execution of the kernel is signal to the host on the parent kernel and it is done by stating of the event associated with the kernel to CL complete. So, just to summarize, you have launched a kernel, you doing that enqueueing operation of the kernel, you have got an event automatically associated to it or you can actually specify which event as it to it.

That event status will be set to CL complete when all of work groups of the kernel and also if there were any child kernel whether all those were most of all those child kernels everything is complete, only then that event associated with the enqueueing of the kernel will be set to CL complete. Now we also have this other function CL waits for events and this waits on the host thread for commands identified by event objects independent list to CL complete.

Event specified in the event list will primarily act as synchronization points. So I believe this should not be there. So just a minor correction. So, when I have CL wait for events this function will wait on the host. So, when I have this function the second last one here wait for events, this is simply I mean suppose you have made it to wait for a set of events that would mean until and unless those event objects will reach this status CL complete individually all the all those events, then only this function job is done otherwise it is still waiting there.

So that is another synchronization primitive. So just to understand using completion of a command, you are attaching or you may possibly attach your event definition to a kernel launch, when that kernel launch and his child kernels have all completed. You get that event being attached, we are being automatically assign the CL complete flag by the runtime system. The other option to synchronize would be a code you have a CL wait for events function and you provide that function with a set of events to synchronize on.


When the runtime system ensures that those set of events reach the status seal complete, then this function will release and lead the execution move forward.

(Refer Slide Time: 20:29)

Command Synchronization

The synchronization points defined in OpenCL include:

- ▶ **Blocking Commands:** Command execution can be blocking or non-blocking. For blocking commands, the OpenCL API functions that enqueue commands don't return until the command has completed. Some of the blocking commands are `clEnqueueReadBuffer`, `clEnqueueWriteBuffer` with `blocking_read` and `blocking_write` set to `CL_TRUE`
- ▶ **Command-queue barrier:** Ensures that all previously enqueued commands to a command-queue have finished execution before any following commands enqueued in the command-queue can begin execution. The OpenCL API functions `clEnqueueBarrierWithWaitList`, `clEnqueueMarkerWithWaitList`.



OpenCL - Runtime Execution Page 2 / 2

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, apart from this, we also have this notion of blocking commands. So these are these are also synchronization points because we have to understand that if a command blocks that means until unless that commands execution is finished, the program straight cannot go forward. So command execution can be blocking or non-blocking. If it is a blocking function, then the events API functions that include commands.

Do not return until a command has completed some of the blocking commands are saying CL enqueue buffer enqueue read buffer and enqueue buffer both of them if they are called with the option blocking read and blocking, we said to CL true. So, as you can see that when you were in making this enqueue commands have read buffer and write buffer like defined earlier, they also had a field where you can set this flag of blocking read and blocking respectively.

And in those cases, they are blocking commands that means if you do not set it, then they will fire a synchronously that means without this read buffer or before finishing the host program and move forward to the next command. Because they will just keep on happening without blocking the actual flow of the execution. But if you said them to blocking, read or blocking, then until unless these commands are complete, the host programs execution will not go forward.

And then you have command queue barriers. They ensured that all previous they include commands to a command queue have finished execution before any following commands

enqueueing a command queue you can begin execution. So, you have the command queue ensures that commands start executing in that order. But if you put a barrier in between it ensures that all the command queue functions we should have started executing in their order they should finish before anything out I mean after the barrier.

Even starts executing the open CL API functions for this cl enqueue barrier with waitlist cl enqueue marker with waitlist. So these are so essentially you are enqueueing a barrier in the command queue. And of course, you can put the barrier with a waitlist that barrier first quits for the events in the waitlist, and then the barrier is active, and then all the commands previous to the barrier are forced to finish before fundal commands can be launched.

(Refer Slide Time: 23:00)

Command Synchronization

The synchronization points defined in OpenCL include:

- ▶ **clFlush**: All previously queued OpenCL commands in `command_queue` are issued to the device associated with `command_queue`. `clFlush` only guarantees that all queued commands to `command_queue` will eventually be submitted to the appropriate device. There is no guarantee that they will be complete after `clFlush` returns.
- ▶ **clFinish**: All previously queued OpenCL commands in `command_queue` are issued to the associated device, and the function blocks until all previously queued commands have completed. `clFinish` does not return until all previously queued commands in `command_queue` have been processed and completed.

OpenCL - Basics Edition | Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

You also have synchronization points defined by `clFlush`. So, you have `clFlush`, then all previously queued open CL commands in a specific command queue or issue to the device associated with the command queue and `clFlush` will guarantee that all these commands in that command queue will eventually be submitted to the appropriate device. And of course, there is no guarantee that they will complete after `Clflush` returns.

So it guarantees that all the commands that in the queues will be submitted to the appropriate device starting from I mean, so you have enqueue them and you have submitted mean cl flushes ensuring that all of them gets submitted to the device for execution. And then you have CL

finish. So here CL finish whatever commands have been queued, they are issued to the host device and the function blocks until all these functions have completed.

So unlike `clFlush`, which is providing you a guarantee that whatever commands have been enqueued earlier, they are all submitted to the device from the command queue CL finishes as if it is executed and it returns successful. It is guaranteeing that whatever commands were enqueued, and I mean, all of them have completed. So this function blocks until all previously queued commands have been completed.

It does not return until all the previous queued commands in the command queue have been processed and completed and so I hope this is the difference that you can understand between `flush` and `finish`.

(Refer Slide Time: 24:42)

Command Synchronization

- ▶ **Command-queue barrier:**
 - ▶ Ensures that all previously queued commands have finished execution and updating memory objects before subsequently enqueued commands begin execution
 - ▶ Can only be used to synchronize between commands in a single command-queue
- ▶ **Waiting on an event**
 - ▶ All OpenCL API functions that enqueue commands return an event that identifies the status of command
 - ▶ Value of the event associated with the command is set to `CL_COMPLETE` when done
- ▶ **`clFinish`:** Blocks until all previously enqueued commands in the command queue have completed

OpenCL - Barrier Synchronization

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And then for the purpose of synchronization, as we discussed that you have command queue barriers and you have worked on an event primitive and command to barrier ensures that all previously queued commands have finished execution and they have updated whatever memory objects or memory objects for commands, which will begin execution afterwards. So, this is the primary reason why you will like to have barriers like the command queue ensures that you start issuing commands in that order.

And if you put the barrier, you can actually ensure that though they finish all the commands that have been include the actually finished and they are before going forward and this can be used for important reasons, for example, as has been specified here that you suppose you need some memory objects to be set up before further command queue execution goes forward and you want some memory objects to be updated and ready by some kernels to have executed earlier and only then you will be executing the next kernel.

This kind of execution we have here you can ensure by using the command queue here barriers. Now the important thing is inside a command queue if you put a barrier it will help you to synchronize between commands inside a single command queue and not across multiple command queues. Now, the other thing is waiting on events all open CL API functions that enqueue commands, they return an event that identifies status of command.


Like we have discussed earlier, that events are returned by the enqueueing command operations only and the events can have values starting from CL I mean starting from I mean waiting, submitted and all that to up to CL complete when the event finishes execution. And finally, you have this CL finish synchronization primitive which blocks until all previous to enqueue commands in the command to have completed like I mean, we have already discussed the will finish and the difference between CL finish and CL flush. So, overall these are the different primitives that will be using.

(Refer Slide Time: 27:03)

Page 2 / 2

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

Now, the next topic would be out of order execution of kernels and memory objects. So far we have assumed that, you have open CL command queue, you have been enqueued them and they can just go and execute in order. And that is not going to hold in general. Maybe we will end the current lecture here and resume from this point in the next lecture. Thank you.