

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

Lecture # 49
OpenCL – Heterogeneous Computing (Contd.)

Hi, welcome back to the lecture series on GPU architecture and programming. So, coming to the continuation of the earlier lecture, I believe we have been talking about multi device programming, where we have built context containing multiple devices, we have set of queue about multiple command queues each one queue for 1 of the device and we are trying to see how the devices can execute kernels concurrently.

(Refer Slide Time: 00:47)

Page 1/1

Multiple Device Programming Work Concurrently Example

- ▶ Multiple devices working in a parallel manner where both GPUs do not use the same buffers and will execute independently.
- ▶ The CPU queue will wait until both GPU devices are finished

(Figure from reference[2])

OpenCL – Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, in that way, this was our example.

(Refer Slide Time: 00:50)

Multiple Device Programming In Pipeline Manner Example

```
// A pipelined model of multidevice execution with single context
// The enqueued kernel on the GPU command queue waits for the kernel on the
// CPU command queue to finish executing
cl_event event0_cpu, event1_gpu;

// Starts as soon as enqueued
err = clEnqueueNDRangeKernel(queue_gpu, kernel1_gpu, 2, NULL, global, local, 0, NULL,
    &event_gpu);

// Starts after event_gpu is on CL_COMPLETE
err = clEnqueueNDRangeKernel(queue_cpu, kernel2_cpu, 2, NULL, global, local, 1, &
    event_gpu, &event_cpu);

// 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
clFlush(queue_cpu);
clFlush(queue_gpu);
```

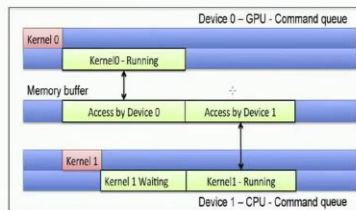


That I have got this program to which we are setting up to the queues, 1 for the CPU and 1 for the GPU. And we are trying to have this kind of we have here that the kernel running on the GPU device who has got some output.

(Refer Slide Time: 01:08)

Multiple Device Programming In Pipeline Manner Example

Multiple devices working in a cooperative manner on the same data such that the CPU queue will wait until the GPU kernel is finished.



(Figure from reference[2])



The kernel running in the CPU GPU device has got some output, and that output is going to be used by the kernel running on the CPU device right. So just a small change here like spurred this model, what we have done is the CPU has device 0, GPU is device 1, I believe that has got exchange. So I hope that can be figured out anyway. So coming to the problem further, so suppose you want this execution and that you enqueue first in the GPU.

So let us say we call device 0 as GPU here although it is a bit different here device 0 CPU, but let us go with the figure here. I believe the concept would be clear so, for device 0, that is the GPU and you want to enqueue the kernel. So this is how you will do it, you include the kernel, but what you do is along with that you have a declared events, events 0 CPU event, 1 GPU, and maybe I believe we will just say we will just say these are events GPU and event CPU.

So this event CPU event GPU. So, when this kernel executes, it will just output and event GPU and all you want is the second kernel should start only when the event GPU is available to it. So it is sensitive to event GPU that is why for the `clEnqueueNDRangeKernel`, you are in queuing this kernel at the point I mean, you have enqueue this kernel for into the CPU queue, queue CPU, and for this kernel, you are waiting for the GPU kernel to finish.

So on when the GPU kernel finishes it outputs the event GPU event. Since I have it in the event list as I make it make these commands sensitive to even GPU. So when this is received, I will have the other kernel which is enqueued in the CPU queue this will start executing, and when this finishes, it will emit this event CPU. Now, then we have these 2 `clflush` commands like discussed earlier to flush both of the queues.

And this only guarantees that all queued commands to this to the command queues they will eventually be submitted to the appropriate device. And what of course there is no guarantee that they will complete after the `clflush` returns right. So this is something we have already discussed that if I reach these are all in a synchronous commands such commands I am getting enqueue here in the host program like this is the host program I am using into enqueue commands.

But once these commands are enqueue when I do a `clflush` of each of the queued the `clflush` is ensuring that whatever commands have enqueue for the CPU whatever commands have enqueued for the GPU, they are dispatched to our devices, whether they are finished or not that I am not ensuring for I will have to put in something else like `clfinish`. So, this is a demonstrative quote to give you the idea that what is the ability here.

So, please mark this correction and also here, just like we are device 0 and device, 1 for the CPU and the GPU, and this is a bit different, so, that will be another here now, without going to the detail less than a week, let us say that how I can make it more generic. For example, I want multiple devices programming working concurrently. And I have more number of devices let us say, I have got 2 devices, 3 devices now, and I have multiple devices working in a parallel manner.

And both devices, both of these GPUs do not use the same left buffer and they will execute independently. And the CPU queue shall wait until both the GPU devices finish right. So let us say that is what I want to happen. So maybe for the idea is that GPU kernel 1 will execute kernel 0 will execute in device 0, it will output something in the memory buffer, then kernel an in parallel kernel, so kernel 0 is executing in device 0 that is, that is the command you have 1 GPU.

Kernel one is executing in some other device, which is device one, which is the command to have another GPU. So as you can see that they are independent kernel, they can execute in parallel. And after they have executed let us say that they copy the data to the memory buffer. And then the requirement is I have a kernel 2 which is supposed to execute in device to here which is a CPU device. And I have enqueue this execution command here in the command queue of this CPU device with a dependency that this kernel should wait.

And it should start executing when the kernel 0 and kernel one both of them have completed their execution, they will then access them and then and then they will output the result.

(Refer Slide Time: 06:34)

Page 8/8

Multiple Device Programming In Concurrent Manner Example

```

// A concurrent and pipelined model of multidevice execution with single
// context on a single platform having 2 GPU and 1 CPU device
// Create 3 command queues, 2 queues for the 2 GPUs and 1 queue for the CPU
// The enqueued kernel on the CPU command queue waits for the kernels on the
// GPU command queues to finish
// Both the GPU devices can execute concurrently as soon as they have their
// respective data as they have no events in their waitlist

cl_event event_gpu[2];
err = clEnqueueNDRangeKernel(queue_gpu_0, kernel_gpu, 2, NULL, global, local, 0, NULL
, &event_gpu[0]);
err = clEnqueueNDRangeKernel(queue_gpu_1, kernel_gpu, 2, NULL, global, local, 0, NULL
, &event_gpu[1]);
// CPU will wait till both GPUs are done executing their kernels
err = clEnqueueNDRangeKernel(queue_cpu, kernel_cpu, 2, NULL, global, local, 0, NULL
, event_gpu, NULL);

clFlush(queue_gpu_0);
clFlush(queue_gpu_1);
clFlush(queue_cpu);

```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, we are trying to see that how that can be done. So, overall in a summary, this is a implementation example of a concurrent and pipeline model of multi device execution, where they have a single context on the platform, but with 2 GPUs and one CPU device. So what we are doing is we are creating three command queues 2 queues for the 2 GPU And one queue for the CPU the kernels are enqueue on the CPU command to the wait for the kernels which enqueued on the 2 GPU command queues that are to finish.

And when both the GPU devices have executed concurrently and they have their and they have done their job they are not only I want the CPU kernel to execute, but these 2 GPU kernels they have no input dependencies so they can just go and executing without any event for which they have to wait so they do not have any waitlist, right. So since they do not have any waitlist, I can have this queue for GPU 0 and Q for GPU one, I can just immediately into the kernels I do not have any waitlist so the event waitlist is null for both of them.

But when this one is done, this kernel GPU kernel this kernels are done for GPU 0 and GPU one. They will be emitting the event GPU 0 and event GPU 1 now the functional they want to implement is that the CPU has should enqueue a kernel in a queue CPU and this kernel should wait for both of these events right. So, that is captured by this event list right. So, I have event GPU this should use waitlist containing 2 of these fields.

So only when both these events go to their complete status, then this kernel which has enqueued here can start executing so kernel CPU can only start executing after kernel GPU CPU in both the devices have executed and this dependency can be captured like this. I do not want any more dependency like there is no further execution based on the completion of the execution of kernel CPU.

So I do not have any further dependency this is set as null. And like earlier, we can just flush all the queue that means these commands are ensuring that all the previous commands from each of these queue have been dispatched. So this is something I will keep on saying this is again a host program, it is just giving that the kernel for execution. It is not forcing them to execute and the queuing is done along with dependencies.

So when the program reaches here, I will wait here until I ensure because it is a cflush is a synchronization primitive like the host program will go past this only when it is ensured that GPU 0 this queue the kernel that has been ensure include here that has been dispatched. So that is ensured then I will wait in cflush queue GPU one unless and until the kernel that was included here that has been dispatched.


And finally I will wait here and I will get first this point only when this kernel which is kernel CPU which is waiting here that gets dispatched that means it has to ensure that the previous kernel has finished and the events are fired. So that this can get a start executing right. So in this way, the cflush actually forcing the execution to happen and with otherwise the host program cannot go beyond this point right. So, with that we have our discussion on multiple device programming with multiple command to support for a stem context.

(Refer Slide Time: 10:17)

Page 8/8

Multiple Command Queues With Different Contexts In Different Devices

- ▶ Context is created with respect to a particular platform
- ▶ For different devices from different platform, we create multiple contexts
- ▶ For separate contexts created for different devices, synchronization using events would not be possible
- ▶ Only way to share data between devices would be to use `clFinish` and then explicitly copy data in and out of a given context and across contexts via host memory space



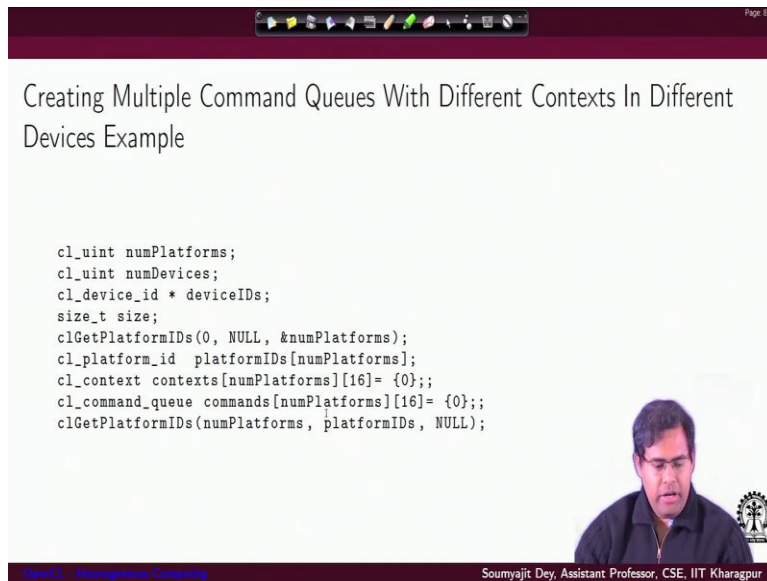
OpenCL - Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, let us talk about a scenario where I have multiple command queues with different contexts in different devices. So, the typical scenario you will want to do is you have context you did with respect to particular platforms, for an in different platforms, you have different devices like that hypothetical example, we were trying to demonstrate earlier. In each of these contexts, you can you have different devices and four different devices, you can create separate contexts and you can synchronize an instance you have now devices sitting in different contexts. You cannot synchronize execution of events across context that is not possible.

So in that case, you have to go through the host program. So the only way to share data between devices would be to use `clfinish`, because if you have `clfinish`, then those command queues have to actually, I mean, I can go beyond the `clfinish` synchronization primitive, only when all the commands that have I have enqueue on up to that point of time in that queue, all of them commit and finish their execution.

Then once that is done, I have the output data of that execution in that device available I have to explicitly, copied from 1 context to the other context. And then inside that context, I can start in giving commands for using that data in this separate context right. So in this way, the idea would be that I have to orchestrate data movement across contexts and I have to use primitives like `clfinish`, to ensure that execution has actually finished in 1 of the context, so that the data is ready, which can be moved across to a different context.

(Refer Slide Time: 12:06)



Creating Multiple Command Queues With Different Contexts In Different Devices Example

```
cl_uint numPlatforms;  
cl_uint numDevices;  
cl_device_id * deviceIDs;  
size_t size;  
clGetPlatformIDs(0, NULL, &numPlatforms);  
cl_platform_id platformIDs[numPlatforms];  
cl_context contexts[numPlatforms][16]= {0};  
cl_command_queue commands[numPlatforms][16]= {0};  
clGetPlatformIDs(numPlatforms, platformIDs, NULL);
```

OpenCL: Heterogeneous Computing
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, if we do look at such examples that you have, you are creating multiple command queues with different contexts that are sitting in different devices, you have different contexts sitting in different devices, and you are creating a different set of multiple queue for the devices. And in this example, as you can see, the typical code would be like you get the platform idea like earlier. And then of course, you should have a context at it and you have a set of queues right now CL command queue array for each of the platforms.

So, as you can see you have a context array and you have a command queue array. So, you have context array for proper platform, right. And you can also have this command queue array as we discussed in the first thing, you will do is you will try to discover the set of platforms right from and their platform IDs.


(Refer Slide Time: 13:06)

Page 8/8

Creating Multiple Command Queues With Different Contexts In Different Devices Example

```
for(int p=0; p < numPlatforms; p++)
{
    errNum = clGetDeviceIDs(platformIDs[i],CL_DEVICE_TYPE_ALL,0,NULL,&numDevices
    );
    deviceIDs = (cl_device_id *)malloc(sizeof(cl_device_id)*numDevices);
    errNum = clGetDeviceIDs(platformIDs[i],CL_DEVICE_TYPE_ALL,numDevices,&
    deviceIDs,NULL);

    for(int d=0; d < numDevices; d++)
    {
        contexts[d] = clCreateContext(NULL, 1, deviceIDs[d], NULL, NULL, &err);
        commands[d] = clCreateCommandQueue(contexts[d],deviceIDs[d],0,0);
    }
}
```



OpenCL - Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And then once you have got the platform IDs, you are trying to get the device IDs. So, this is the kind of code we are a bit familiar with. So, inside the loop you are looping inside the number of platforms. For each platform you get inside the loop, you identify all the devices in the platform using this command. As you can see that if you are using CL get device IDs with the flag, CL device type all is discovering all the IDs all the all the devices have in total number of devices and that is pushed into this non device variable.

And then you allocate suitable amount of memory which is equal to the number of devices multiplied by the size of CL device ID you look at that must be able to this device IDs at it. And then again, you give a call to CL get device IDs. So that for this specific platform, we are inside a loop in each iteration, we are inside 1 platform. So for that platform, all the devices, ideas that have been discovered, I mean, they are actually provided in into this device ID array right.

Now, for all of these devices together, what you can do is, you run this kind of loop using this loop, you are creating a set of contexts, separate context for each device as you can see. So, technically what we are doing is we have a set of platforms. For each platform, you discover the set of devices for each device, which is whose ID is stored in this device IDs and this is done in the last line for each device, you store its IDs in the device IDs array. For each of the device IDs, you create one context for each of the device IDs inside each context, you create a command queue right so these are you move forward.

So this is again, this is an example problem, we are trying to show examples of pipeline execution of multiple kernels in different command queues in this case. So what we have done is we have defined 2 contexts for 2 devices. And each of these devices have their own command q. So let us say this is one platform, one of the platforms inside this platform, I have 2 devices, D1 D2. For each of these devices, I have created a context or better to sit here and I find and for each of these I have inside each of these contexts these devices, and they are command queues, right.

So I would say I have a queue 1 and I have a queue 2. This is the kind of setup we are looking at. So now considered that I want to do execute things in such a way that there is a dependency between kernel 1 and kernel 2 that is output of kernel 1 is user input to kernel 2 what we do is we have a CPU and a GPU in the platform. And kernel I assign kernel 1 to the CPU and I send kernel 2 to the GPU, right. So for doing this whatever code is required, let us say that has been already written here.

And in that way, and I have set up both of these queues. One is the queue for the CPU, which is one of the devices one is the queue for the GPU use other device and they have been assigned the task executing kernel 1 and kernel 2. So, first I do a write buffer. So, in the queue of CPU, I do a write of some buffer A and then in the queue of the CPU, I do a write of the other buffer B and then I execute kernel one in the CPU which will be working on the data of buffer A and buffer B.

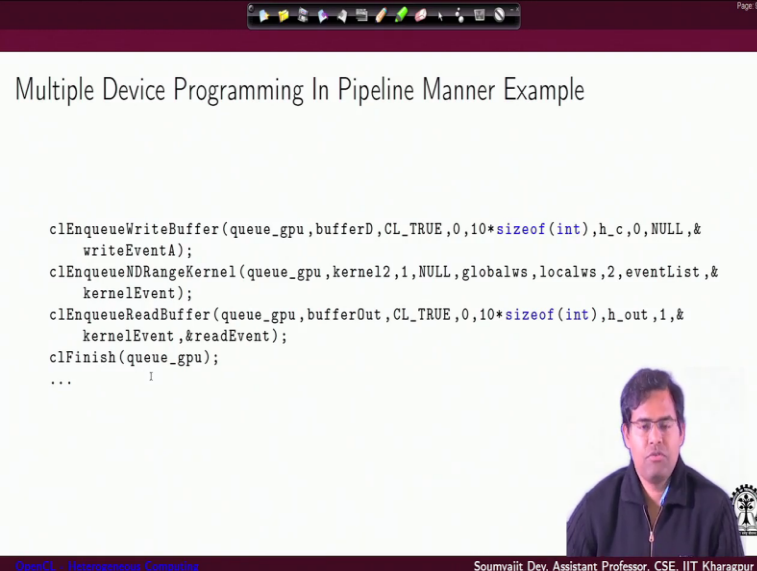
And once it is done, it will output this kernel event, now, I have this read buffer command here. So that is a way I can ensure that well, this has then this command which I enqueue for a launching a kernel which is kernel alone, and that has been finished when this kernel event shall here, and this is in the event list for speed buffer. So only when this kernel has been actually launched from the queue to the device, the CPU device and it has finished execution, I will have these event ready and with this event firing now I have the read buffer command read execute.

And with that read buffer, I will have the buffer see which is the output of kernel one it will be transferred from the it will be transferred and once it is transferred, I will have the read event this read event set right and at the end I have this CL finish here if I go with this the host program

goes beyond this point. That means, all these commands that have queued up in the CPU queue, they are actually dispersed even though they are not finished execution.

So, here I am blocking it until all previously queued commands this is `clfinish` or `clflush`. So, I am actually blocked here until then unless all these commands which has been queued up they have been dispersed as well as all of them finish, right? Because is `clfinished`, there is not a `clflush`. So, they have their finished execution in their stated device.

(Refer Slide Time: 13:06)



The slide displays the following OpenCL code:

```
clEnqueueWriteBuffer(queue_gpu,bufferD,CL_TRUE,0,10*sizeof(int),h_c,0,NULL,&
writeEventA);
clEnqueueNDRangeKernel(queue_gpu,kernel2,1,NULL,globalws,localws,2,eventList,&
kernelEvent);
clEnqueueReadBuffer(queue_gpu,bufferOut,CL_TRUE,0,10*sizeof(int),h_out,1,&
kernelEvent,&readEvent);
clFinish(queue_gpu);
...
```

The slide also features a video feed of the speaker, Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur, in the bottom right corner. The footer of the slide reads "OpenCL - Heterogeneous Computing" and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

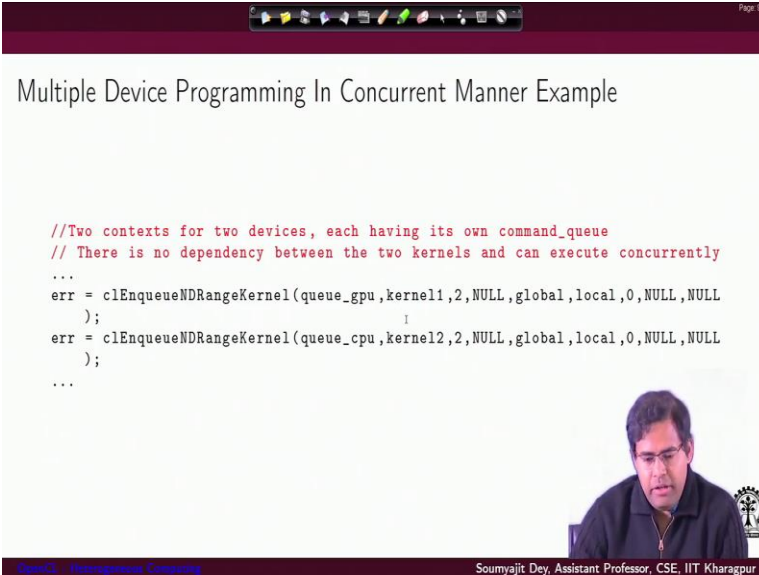
So, all that is happening is almost same. Only thing is here we are showing that we are working with different contexts, right, we have this kernel data, which is there in different contexts. I mean, since they are in different contexts, so I have to make these reads and writes. So I put the data in the context of the CPU, right? And then, and so for that, I have to enqueue commands. And then I execute the kernel, after executing the kernel, have to treat it back, because it is not in there. I mean, the GPU device is not in the context.

And then I have to read back the data. From I have to read back this the data to the context of the GPU. So, for that, I will now again have this right buffer command executing. So, it will copy some buffer d to the queue of the GPU and then I will have kernel 2 executing in the GPU. So, as you can see that this is going to execute only when so, I have this kernel to which is executing

here. Now, for this I have a guarantee the since I have put in a clfinish so, all these operations that were done earlier, they are finished.

So, after that I have this kernel 2 executing and once I have content to complete it, which is being ensured by the event kernel event, which is there in the sensitivity list of this command for this read buffer command. So since so only when kernel event fires, that means this come this is done, then only I am going to read back the output. Assume that is there in some buffer out. I am going to read it back. And then I give again, I fire the read event command. So again, I will put in a clfinish in the queue GPU.

(Refer Slide Time: 21:23)



The slide displays a code snippet for concurrent execution on two devices. The code uses `clEnqueueNDRangeKernel` to enqueue two kernels, one on the GPU and one on the CPU, with no dependencies between them. The code is as follows:

```
//Two contexts for two devices, each having its own command_queue
// There is no dependency between the two kernels and can execute concurrently
...
err = clEnqueueNDRangeKernel(queue_gpu, kernel1, 2, NULL, global, local, 0, NULL, NULL
    );
err = clEnqueueNDRangeKernel(queue_cpu, kernel2, 2, NULL, global, local, 0, NULL, NULL
    );
...
```

The slide also features a video feed of the presenter, Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur, in the bottom right corner. The slide title is "Multiple Device Programming In Concurrent Manner Example".

So I have this 2 context for the 2 devices, each of them have their own command queues. And there is no dependency between the 2 kernels and they can execute completely concurrently. So if I want that kind of behavior, then I would just enqueue them in parallel without any kind of dependency coming back here. So if I have this line of the code, the execution is pipeline. That means kernels which is executing in the CPU, once it is done.

(Refer Slide Time: 22:04)


Page 8/8

Multiple Device Programming In Pipeline Manner Example

```

//Two contexts for two devices, each having its own command_queue
//There is a dependency between kernel 1 and kernel 2, output of kernel1 is
used as input to kernel2
//Kernel1 is assigned to CPU and kernel 2 is assigned to GPU
...
clEnqueueWriteBuffer(queue_cpu,bufferA,CL_TRUE,0,10*sizeof(int),h_a,0,NULL,&
writeEventA);
clEnqueueWriteBuffer(queue_cpu,bufferB,CL_TRUE,0,10*sizeof(int),h_b,0,NULL,&
writeEventB);
clEnqueueNDRangeKernel(queue_cpu,kernel1,1,NULL,globalws,localws,2,eventList,&
kernelEvent);
clEnqueueReadBuffer(queue_cpu,bufferC,CL_TRUE,0,10*sizeof(int),h_c,1,&
kernelEvent,&readEvent);
clFinish(queue_cpu);
//Blocks until all previously queued OpenCL commands in a command-que
issued to the associated device and have completed.
...

```



OpenCL - Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, this is the other kernel is executing in the CPU. The read commands have been the right commands actually ensure that buffer A and buffer B we are copied in the context of the CPU and then the kernel execution of CPU and then the output of the kernel which is buffer C is copied from the context of the CPU to the host side and then once I have copied this, so these are the host side data points maybe we were not showing how they are initialized assume that h a, h b or h c are available in the host side memory.

They have been declared null of h a, h b and h c has been initialized. So, we copy the content of h a and h b into buffer a and buffer b which are in the context of the CPU device, execute the kernel and then after the kernel executes, I have the data available. So, I have a read buffer now, which is ensuring that the kernel executed through this event dependency of kernel event and then it reads back and it reads back the content of buffer C to the host side memory h c.

And then this is being written to the there is a right buffer command, which is writing h c in some buffer D, which is there in the context of the GPU. So, maybe if we just repeat what things are going on so, I have these 2 contexts they are they are separate contexts. That is why we are doing all this reads and writes. So, you have CPU, you have the queue for that we have in queue in commands, you have the GPU, you have just put them in separate contexts, right.

And we are trying to see that since they are in separate contexts, I have to set up the offers the buffer A and B, where I copy the data from the host side h underscore A and underscore B they will the kernel will execute it will have the data in the buffer C from that I will copy back to h c host side memory then I will be copying h c to the GPU side buffer D and now I will have this ND range kernel GPU which is going to start execution and once it has executed you will see that it will output again another kernel event.

So, it will again output and kernel event here and this kernel event is actually being used for ensuring that now when to start that I will do the read buffer again So, This output is available here in some buffer out the buff output of this kernel two is available in some buffer out and then it is copied back to the host side memory which is h out right. So, these are things are happening and finally, we have a clfinish to flush out both this queue and the other queue here.

So, this is how we are trying to show that how things are pipeline that means, when the execution is done, we are able to actually copy things to the next device. Of course, when and how this is useful. That is something that you have to figure out where then is at all useful for the application. We are just trying to show that how things can be done that how you can have device executing in one context and you can get output from that context and you can copy it to the other context.


We are trying to show that it is necessary when you have devices sitting in different contexts, then the memory objects are not shared. So, whatever is the output of a corner executing inside 1 context which is to be copied to the host side memory and it has to be copied back to the other context. Now, of course, if there are no dependencies here, you but as you can just enqueue stuff in 2 different contexts accused in 2 different contexts and they can just continue execution right.

(Refer Slide Time: 26:34)

Page 10/10

Concurrent Kernel Execution

- ▶ Concurrency is property of a system in which a set of tasks in a system can remain active and make progress at the same time
- ▶ Programmers need to identify the concurrency in their problem and efficiently schedule in the host program
- ▶ The concurrent tasks can be running-
 - ▶ Different kernels from different independent applications
 - ▶ Different kernels without dependency between them from same application
 - ▶ Partitioned instances of same kernel that are SIMD in nature



OpenCL - Heterogeneous Computing Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, overall, if we are talking about such concurrent control execution, so then this concordance is a property of a system in which you have a set of tasks and they can remain active and make progress. So that has seen at the same time, the programmer would need to identify the concurrency in their problem in case you are you want to develop a program for such concurrently execution in heterogeneous platform. And if the point is the program id up to remember the differences.

The abstractions and accordingly do the mapping of the workloads or the kernels and their dependencies into different devices with through the abstraction of device kernels and platforms. And also the programmer has to use this idea of events and their synchronizations to efficiently schedule the host program. And the concurrent tasks that may be running will be of the following kind, you can have different kernels for different independent applications, you can have different kernels without any kind of dependency between them.

They are from the same application that is also possible right to have 2 dependent kernel 2 independent kernel running maybe on they are both of their outputs, something else can depend. But these 2 kernels have no dependencies, there is also part possible. So this is also when you can have concurrency of tasks or kernels and you can have situation the 1 kernel, it has been partition that is let to the kernel requires 1024 threads to execute.

You just partition the input data space to 2 different devices, you copy it in a partition way to 2 different devices. And you launch half of the number of threads into device 1 how the number of threads in the device to and execute them. So that is also a partition instance of the same kernel and this also like concurrent tasks execution. So these are the possibilities in which you have concurrent all execution in a multi device platform with this, we like to end this lecture. Thank you.