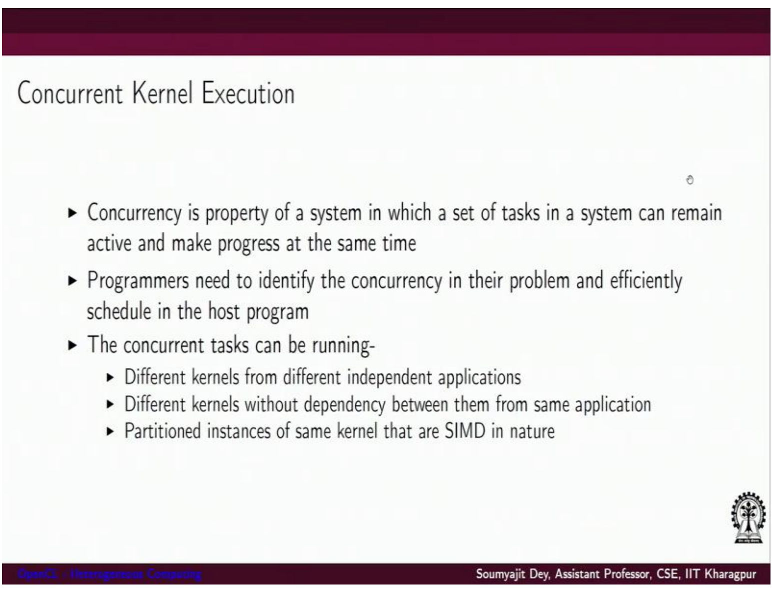
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 50
OpenCL – Heterogeneous Computing (Contd.,)

Hi, welcome back to the lecture series on GPU architectures and programming. So, I believe in the last lecture, we have been some are kind of discussing more around concurrent kernel execution and how concurrent kernels can be map.

(Refer Slide Time: 00:38)

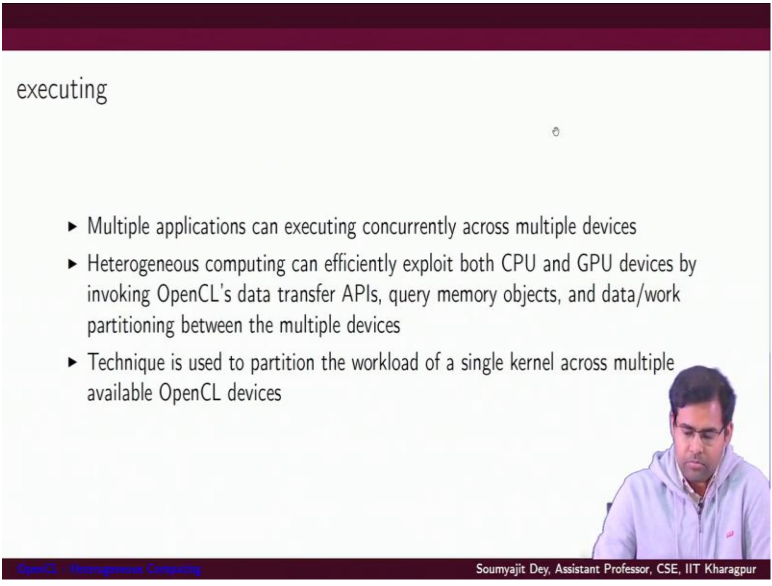


To an underlying OpenCL device. So, in that context we talked about the falling possible options like if you have concurrent tasks, then 1 option can be that you have different these tasks are all different kernels which are coming from different independent application so they do not have any dependency and so, also the other option is that will there is a same kernel, but multiple paths will the same application, but from the same application multiple kernels have been launched.

And they do not have really any dependency among each other. So then you have them, you can just run them concurrently across multiple devices. Also, suppose you have 1 kernel and you want to exploit more compute capabilities of the hardware another option can be that you partition the kernel across multiple devices and execute them. So we will fine will I mean, there

has to be some motivation like while why you will want to do that. We will discuss on the motivation.

(Refer Slide Time: 01:42)



So, in general, you can have as we are discussing that you have this multiple kernels or multiple kernels of the same application are they belonging to different applications they are executing concurrently but in case you want to increase the concurrency more 1 option is in case you have also available devices, you partition the kernel across devices. But the issue is how can it really be done.

First of all, the good thing about doing it is that you can exploit both CPU and GPU devices by using both the devices to kind of execute the same kernel. And this is 1 advantage that OpenCL has, right because an OpenCL kernel does not make a distinction between CPU and GPU devices; it is built in for heterogeneous computing. So you can actually split the kernel into two sister kernels, I would say and dispatch them to multiple devices.

And I mean, of course, the 2 devices have to be available in the system for doing that, but assuming they are available, then what is happening is you are exploiting more amount of parallelism in that way. And this can really be done using OpenCL returns for APIs query or I mean memory objects and other and support for multiple devices that is available.

(Refer Slide Time: 03:11)

Partitioning

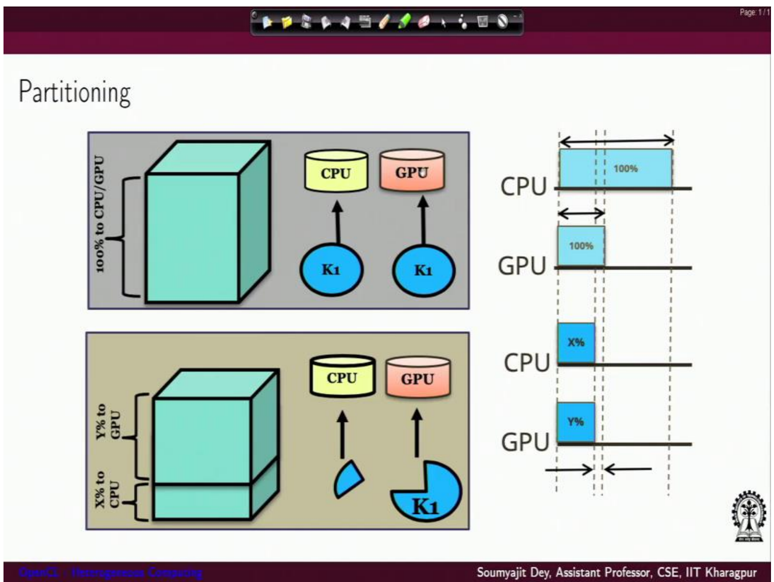
- Some kernels performs better on either CPU or GPU devices
- While executing a kernel on a specific device the other available devices may remain idle
- Partitioning is a technique used to partition the data efficiently and distribute them across multiple OpenCL devices
- Then launch same kernel with partitioned data across multiple OpenCL de
- The partitioned kernels can run concurrently on different devices reducing execution time

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, to motivate a bit more, in general, you can have kernels, which are GPU friendly kernels, I would say, because let us say the kernel has too much of parallelism. And inside each parallel thread, there is not much of control oriented dependency. So then it is a very much GPU oriented kernel. And it makes sense to execute in the GPU whereas you may have an OpenCL kernel with lot of branches, divergences and control dependencies inside the kernel.

So that will make up case for CPU only execution. So overall, you have a design space, like whether you want to execute a kernel fully in the CPU, whether you want to execute the kernel across CPU and GPU or whether you want to execute the kernel fully in a GPU. So, just to create a problem space here this is what we can do like suppose.

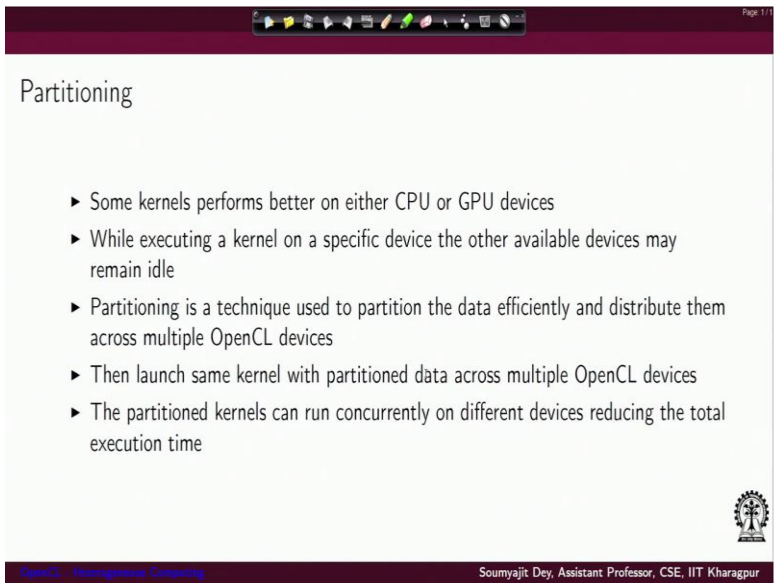
(Refer Slide Time: 04:20)



You have this kind of kernel and you are trying to submit and this is the full job, there is a full data space on which this kernel K 1 is going to work. So, it is essentially defining a transformation which will be carried on the internet data space. You can dispatch that kernel to the CPU and get the work done on this data service. You can dispatch it on the GPU, you can get the job done on the on this full data space. Or you can split this data space to individual memory segments.

And get to Kernel 1, Kernel dispatches to CPU 1 Kernel dispatches to GPU and then you have both of these devices engaged in parallel for doing the job, which is a good thing right if you disperse it only since CPU the GPU is idle plus in case there is no other application or some other kernels application some other kernels which are running here, right. But in case it this is free, there is no other kernel which is looking to execute here there, why not to split it like this and execute some part on the CPU and some part of the GPU.

(Refer Slide Time: 05:29)



So, we will first try to figure out when and how these things make sense. So, there is a primary motivation that it may happen that some other devices are available, they are idle, so, it makes sense to partition the data space and dispatch multiple kernels on the partitions data space across engaging with multiple OpenCL devices. And effectively what you are doing is you are kind of exploiting the systems available concurrency more efficiently. But let us also discuss when and how this thing makes sense.

So suppose you have this data space. And you have. So let us say this is the host memory and since it is attached to the CPU device, so right. So, you are bringing the data to the CPU cache and then you are executing it on the CPU pipeline here. And then you have an off chip connection to the bus. And on this bus, you have a connection to the GPU memory, let us say that device memory and the device in general. So this is like the entire device. There may be multi devices.

So what is the overhead of actually partitioning the data space and executing in the CPU, or I mean across the CPU and the GPU. So, if you do a CPU only execution, there is no data transfer involved, right? So you will launch as many I mean, you are OpenCL Kernel will launch as many threads as the number of total number of parallel assembly operations supported by your CPU. And those threads will actually execute in parallel, and they will compute on these data service.

So, if it is a I mean, brand heavy Kernel, I would say, then make sense to have it in the CPU by the computational nature, its CPU friendly kernel. So then it is makes sense not to send it to the GPU. Why because first of all, GPU are not very efficient. They do not have speculative execution and other supports that you have in CPUs. So, for handling branches, the CPU as a device is much more efficient.

Now, when you are thinking of executing on the GPU side, then you have the problem, which is that this is then you have the host program which is executing on the CPU, this host program has to first copy the data from is from the host side memory, which is this 1 will have to copy the data from this side memory to and to the GPUs DRAM which is here right. So, there is some additional data transfer over it that you have.

So, from the memory that is referenced by the CPU, you have to first you have to first define buffers in this memory and you have to copy that data into this memory. Once the input buffers are set up, then you are going to launch the kernel right. So, let us say that CPU only execution time for a kernel is exCPU. The GPU only execution time for the kernel is a exGpu and let us for the sake of generality assume that in total you have 100 threads.

And if you are working with x number of threads, let us say a factor of x, then the execution time in the CPU would be x multiplied by ex CPU and the rest of the threads. So let us say you have 100 - x threads, which are executing in the ex GPU assuming that you are splitting this number of threads in parallel across the devices question is when will I do it? I have I will leave now the issue is there is this data transfer time right.

So, this factor x have to choose in such a way that the execution time it is multiple I mean for x number of threads in the CPU should actually equal the execution of time of the rest of the threads in the GPU plus the data transfer overhead of this 100 - of the requisite memory elements for this 100 - x threats, right because you have to send these many elements to the GPU memory and you want to do the compute, right.

So then I would be able to say that let us say the origin or execution time is this or in this GPU it is this but when I partition it, the execution time becomes equal to this x multiplied by this ex CPU, right. So just a small correction, there is 2 here. So, if we are saying that x is a factor, so it is like 1 - x. So, you multiply it by 100 here, and you multiply it by 100 here, right. So let us say x is 0.2 that means you are executing 20 threads in the CPU and the rest of the 80 threads in the GPU.

And you are in case it happens that executing 20 threads in the CPU takes same time as a time required to copy the 80 threads corresponding data elements to the GPU copied back and also do the operations then this is execution from that is required right. Now, it may so happen that the data transfer for it is too high. For some kernel, in that case, this is not a good option right. Because if the data transfer of it I mean, outwits whatever benefit you can have to parallel execution.

Then this does not make sense you will only like to have it we do not like to be here if the available parallelism for whatever threats you are now launching in the GPU is much more with just the 2 is over it right. Otherwise this kind of execution will not make sense. But this is for the kind of devices where we have a discrete GPU. So let us say you have an Intel CPU, an Nvidia GPU or some in AMD GPU, let some other vendors GPU use there, they are connecting through a PCI Express Bus.

So both are separate chips, right. Now, in that case, this was execution timing, I mean some possible in these are all very, we are making a lot of assumption here. We are saying that the execution time is proportional with the number of threads and all that and so and also since this is a factor, so and this is the execution time for 100 threads, so we can remove it is just x multiplied by this and 1 - x multiplied by this and, but as we are saying that for integrated GPU, this execution model is not going to hold.

Because here we actually capture the data transfer Raiders for over it. Now the question is what is an integrated GPU? Well, you have modern CPUs and come I mean other I mean devices from different vendors, we are on the same die you can have a CPU device and a GPU device and also

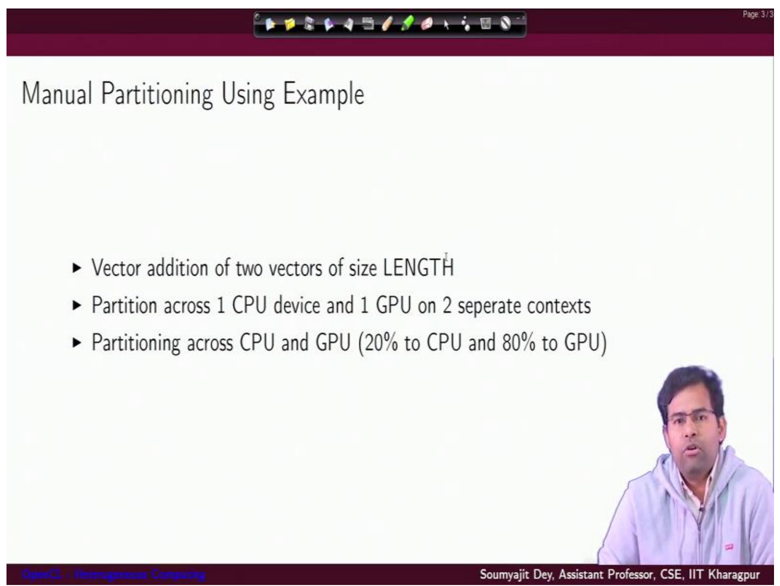
shared memory right. So, there is no data transfer over it when you are defining buffers for this class of devices.

So, when you partition corners for this class of devices, that data server is not there. So the calculations have to be done in a different way. So all these points to the idea that maybe in certain cases, it may sense that you really partition the data space and launch multiple kernels, in case you have multiple devices, which are freely available in the system. So with this motivation, if we go forward here, and let us see how really this can be done.

So in this classic example, what we are really trying to say is, let us break it 1 into 2 parts. So, you split that in a way that x number of CPU threads go to CPU y percent of threads go to GPU, total is 100 here. And accordingly, you create 2 different kernels, 1 for the CPU and one for the GPU. And let us say for the system, the CPU only execution time is this the GPU only execution time is this but if split an execute, since they have some parallelism, they will execute in a shorter amount of time.

So here, the overall execution time was this because up to this point, the whole job is not done and GPU overall execution time was this so this is like a picture for a GPU friendly kernel. That is why I make a small split, just to load balances since anywhere GPU friendly Kernel, I sent smaller number of threads to the CPU and more number of threads to the GPU. So, that after this much time, both of them are getting done.

(Refer Slide Time: 15:19)



So let us take an example of such a piece of code, considered this idea of how I mean you can I mean as a programmer, partition a kernel into 2 kernels for execution in parallel. Of course, you can write a tool which will automate this transformation, you can do a write a compiler kind of tool, which will take as input a kernel and take a partitioning factor and create multiple kernels for multiple devices which can execute in parallel.

Now, 1 important thing I would like to say that when we are assuming we are doing this kind of partitioning, we are, we are also thinking that there is no dependency among data in the data space. So when I launch a thread where we are, thinking that the kernels are such that the thread does per unit data computation. So, the Ith thread, let us say is working on elements from Ai and Bi in the input buffers.

So the IS threat is not accessing let us say, Ai - 100, that will actually complicate the speeding process, right. So, of course, in that case, also you can write a split code. But in our examples, we are making that simplistic assumption, just to motivate the idea here. So, here we will show an example. Where do you do vector addition of 2 vectors, 2 sizes of length equal to some constant, and you partition across 1 CPU device and 1 GPU device, you create 2 separate contexts, 1 for each device, and second 20% of our threads to the CPU and 80% of the third GPU.

(Refer Slide Time: 16:53)

Manual Partitioning Using Example

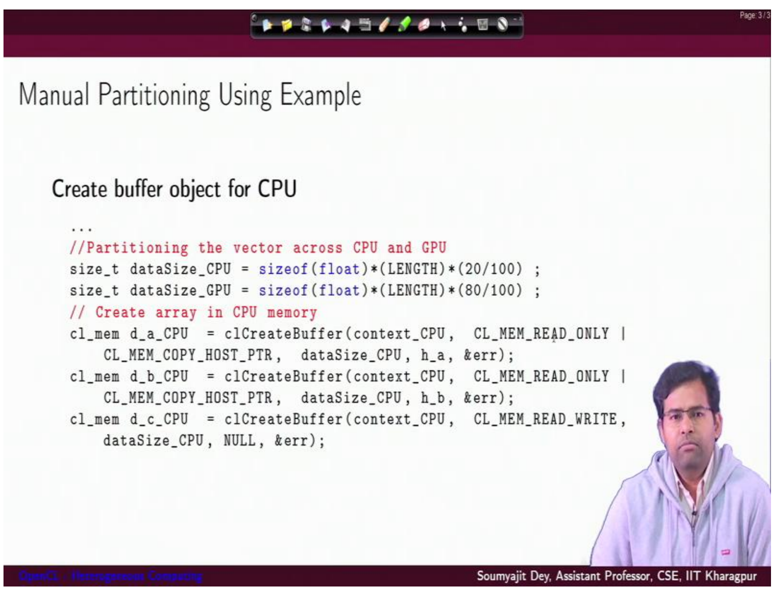
Initialisation and declaration host data

```
...
size_t dataSize = sizeof ( float ) * LENGTH ; //LENGTH is size of vector
float * h_a = ( float *) malloc ( dataSize ); // a vector
float * h_b = ( float *) malloc ( dataSize ); // b vector
float * h_c = ( float *) malloc ( dataSize ); // c vector ( result )
cl_int err ; // error code returned from OpenCL call
// Fill vectors a and b with random float values
int i = 0;
for ( int i = 0; i < LENGTH ; i ++) {
  h_a [i] = rand () / ( float ) RAND_MAX ;
  h_b [i] = rand () / ( float ) RAND_MAX ;
}
...
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, initially, I mean these are simple code, right? Which is just that you are just by doing a partitioning of data space executing across devices. So, you have a data size and corresponding to that you initialize 3 vectors in the host or 3 buffer for storing the input vectors in the host memory right and also the output vector space. Now so, you initialize the 2 input vectors, h a and h b.

(Refer Slide Time: 17:22)



And then you create suitable data spaces for I mean for partitioning, what you do is you calculate what is the data size for the CPU and for the data size of the GPU. So, assuming we are doing an 80 and 20 split, then you need to create suitable buffers in the CPU memory. So, what we do is, you create a using CL create buffer, you create a buffer in the CPU memory inside the context of

the CPU right. So, you create in the context of the CPU. So, here again we are assuming that all these devices have been nicely set up.

And you are just having 2 context setup one for a CPU device and one for a GPU device right. Also we are assuming that the host program is not running in any of them that is why we will actually be transferring data to the CPU and the GPU both right. So you do a CL create buffer two and four using which you create a buffer in the CPU for storing data of the size of the partition that we have defined is the size of the partition and the CPU. Similarly, to the size of the partition in the GPU, you create a buffer in the GPU side right.

So these are the 3 calls, which are all on the CPU side for creating 3 input 3 buffers for the CPU, 2 of the input buffers, and 1 for the writing the output data in the CPU, right using CL create buffer. As you can see, for all of them, the context you CPU and buffer sizes, data size which is calculated here. And since these are input buffers, so they are in read only form.

(Refer Slide Time: 19:07)

Manual Partitioning Example

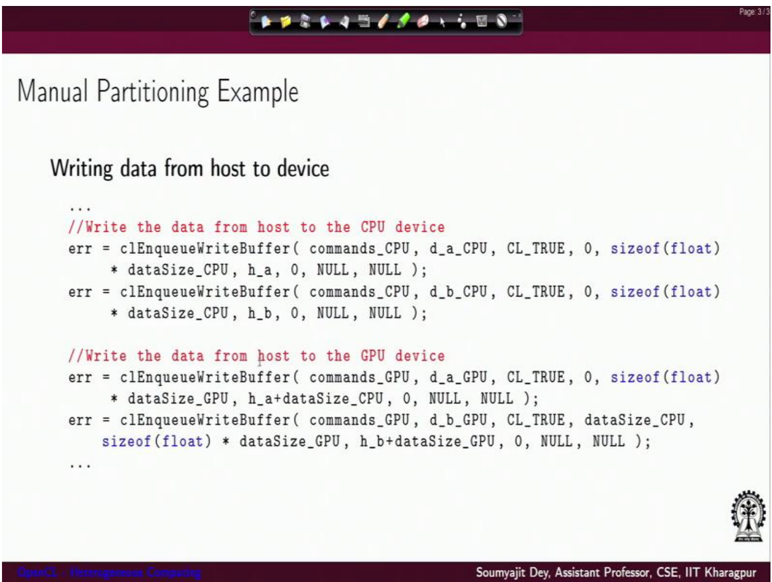
Create buffer object for GPU

```
// Create array in GPU memory
cl_mem d_a_GPU  = clCreateBuffer(context_GPU,  CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR,  dataSize_GPU, h_a, &err);
cl_mem d_b_GPU  = clCreateBuffer(context_GPU,  CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR,  dataSize_GPU, h_b, &err);
cl_mem d_c_GPU  = clCreateBuffer(context_GPU,  CL_MEM_READ_WRITE,
    dataSize_GPU, NULL, &err);
...
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And once this is done, you also do a same thing on the GPU side GPU device, again will just repeat we are assuming the host programming is not executing in any of these devices. That is why you have to create buffers in both of these devices. So you in order to set up the problems for the GPU device, you also create the input buffers and output buffer where it will write the result.

(Refer Slide Time: 19:30)

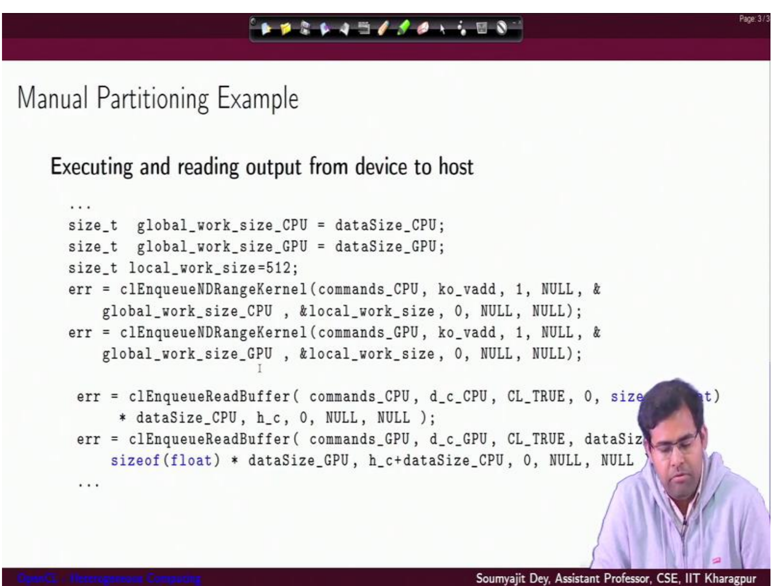


And then you do the data transfer of data size underscore CPU amount into the CPU device. And this is the command to the CPU command CPU right. So you set up the 2 input, the input vectors, input buffers of size. So these are the buffers d a CPU and d b GPU. I mean, they both are the input buffers d a CPU and d b CPU, which are of size data size CPU, right. And they are essentially taking values from h a and h b which are the original input that is sitting in the host side memory.

And then you do the thing that you just do similar write buffers for the GPU device. So again from the host side memory, you copy data of size data size GPU to the GPU device. As you can see, you are copying both from h a and h b, but you are copying from an offset right because starting from the h a address you have already copied data up to size data size CPU. So now you copy with an offset of from h a + data size CPU, right and the amount of data you copies data size GPU, right.

So this is how you do it so you execute and essentially you are enqueuing this write buffer commands. So you have enqueue these 2 right buffer commands here and to the CPU device we have also include to right before comments to the GPU device.

(Refer Slide Time: 21:03)



Once this is done, you have to execute and read the outputs from then I mean devices to the host. So first you execute the kernels. So you have to execute the kernels in the comment queues of I mean f by enqueuing suitable Kernel launch commands in both the commands queues of the commands CPU as well as the commands GPU. So as you can see, we have 2 Enqueue ND range kernel calls, one for the command CPU queue, and then for the commands GPU queue.

So once this is done, then the host program we like to copy that the results from the 2 output queue output buffers that have been set up in the CPU and the GPU device. So those are the buffers. This is CPU and this is GPU, so you copy them back again, by enqueuing suitable read buffer comments into the individual comment queues of the CPU and GPU device.

(Refer Slide Time: 21:58)

Manual Partitioning Example

Checking output

```
//Synchronize
clFlush(commands_CPU);
clFlush(commands_CPU);

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

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

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So once that is done, you will want to synchronize and flush with the command queue of both the devices right. So, just to ensure that all the comments have been dispatched for both the devices and finally is the results. So, that would be our idea of doing the partitioning of an OpenCL program. And as you can see that we have already explained that what is the situation in which you have some advantage?

First of all the data transfer time should be such that it is not too heavy, but you have that you can you are doing enough work in the GPU in parallel to either transport time and overall you have an advantage by exploiting the parallelism. So, that also depends on the style of the kernel. So, what is the nature of the kernel supports this part I mean is such that partitioning gives you an advantage and you have multiple devices available.

Then it will make sense to do the partitioning. Also, as we said it may happen that you have devices let us say you have several devices like modern mobile associates, where you have associates with 1 chip CPUs and GPUs, multiple CPU devices, multiple GPU devices available which shared memory was interference with shared memory interfaces. So that they can just I mean copy data quickly across a shared memory,

I mean, they can produce the outputs and right to a shared memory, right. So there is no question of data transfer reference for it for such kind of devices. And they are partitioning also does not

have too much overhead with respect to the data transfer. So depending on what kind of device you choose, you have to select whether we have to first decide whether partitioning is actually helpful or is it the kernel is the kernel is very much CPU friendly, or very much GPU friendly, I would say that it does not make sense to partition.

There are kernels with different kinds of characteristics. So you have to understand the characteristics of your kernel, whether it is a branch checking Kernel or compute heavy and parallel Kernel and also, whether there is enough number of independent processing for each kernel thread, so that the data space can be split nicely and dispatched across devices. So, you gain you may gain depending on the kernel characteristics by exploiting the parallelism provided you have multiple devices available. So, that is the advantage of partition execution of OpenCL programs. And with this we will like to end this lecture. Thank you.