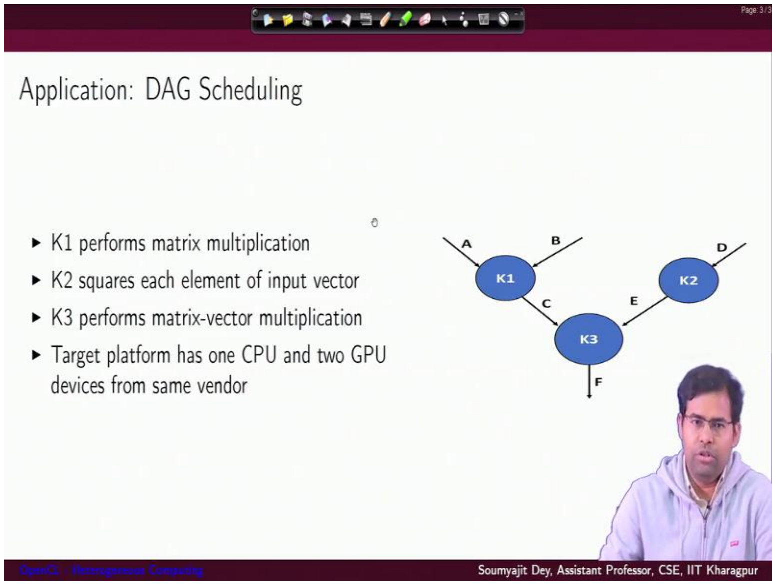
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology-Kharagpur

Lecture # 51
OpenCL – Heterogeneous Computing (Contd.,)

Hi, welcome back to the lecture series on GPU architectures and programming. In the last lecture we have covered how OpenCL kernels can be partitioned across multiple devices and executed in case and when it is advantages and how that helps in terms of gaining more exploiting more concurrent execution. And we have also understood that this is a facility available with OpenCL because the OpenCL kernels are common in case the vendor provides OpenCL implementation for any device. The same kernel code can be mapped through different possible devices provided you have suitable OpenCL libraries and devices.

(Refer Slide Time: 01:00)



Application: DAG Scheduling

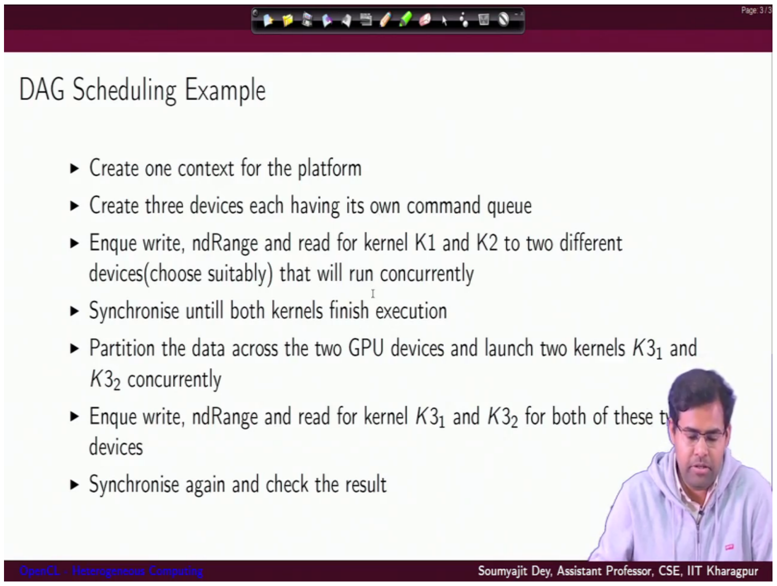
- K1 performs matrix multiplication
- K2 squares each element of input vector
- K3 performs matrix-vector multiplication
- Target platform has one CPU and two GPU devices from same vendor

A
B
D
K1
K2
C
E
K3
F
Page 3/3
OpenCL – Heterogeneous Computing
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

The Next topic we like to touch in is how to use this concept in a general case where you have a wearable object application DAG for directed a cyclic graph. And you want to execute this kind of an application on a system architecture company. I mean, where do you have multiple devices. So what we are assuming in this small example is said this is a task graph. So you have 3 input buffers, A, B, and D. So you have 2, kernels who can execute initially in parallel, let us say K1 and K2, K1 let us say this again, an example performance metrics multiplication, K 2 takes 1 input buffer, all for all the elements. It performs square computation for each of the elements.

And they produce their outputs in buffer, C, and E. And these are basically buffers which are input buffers for the kernel K3, and this kernel will live in performer a matrix vector multiplication. Let us say this is a workload for us. And we want to execute this workload on a system where you have a CPU and 2 GPU devices coming from the same vendor. So, I mean, so, you can set up a context with this multiple devices and set up OpenCL queues for each of these devices.

(Refer Slide Time: 02:20)



DAG Scheduling Example

- Create one context for the platform
- Create three devices each having its own command queue
- Enque write, ndRange and read for kernel K1 and K2 to two different devices(choose suitably) that will run concurrently
- Synchronise untill both kernels finish execution
- Partition the data across the two GPU devices and launch two kernels $K3_1$ and $K3_2$ concurrently
- Enque write, ndRange and read for kernel $K3_1$ and $K3_2$ for both of these t devices
- Synchronise again and check the result

OpenCL - Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, essentially, we will provide a host program, which will create the context for the platform, it will create 3d OpenCL devices, each having their own command queues. And then you can enque the write ndRange and read operations for kernel K 1 and K 2 on 2 different devices that will run concurrently. Now you have to choose that suitably maybe you have a programmer, understand that which kernel is more compute intensive, it has less divergence and all that and accordingly, you can make choices, right?

So you can map this K 1 and K 2, 2 of the devices and you synchronize until both the kernels finished execution. So essentially word in the host code once both of the kernels finished execution, then you may want to do just an example mapping were talking about that you may want to map this data space, this input data space, which is C and E into 2 different GPU devices and launch 2 kernels K 3 1 and K 2 concurrently.

So what we are suggesting is, so here, you have the buffer C, full of data produced by K1 and the buffer E produced by data produced by K 2, right. So maybe we are again talking about a possible mapping. Let us say you have GPU device 1 and GPU device 2 you can copy a part of this here right a part of this here, part of this here, part of this here and then you can launch a kernel K 3 1 here.

You can launch a kernel K 3 2 here produced 2 outputs and then you copy to the host side memory right copy these 2 host side memory to get the final output you may want to do that in because you have multiple search devices available and in case this data transfer, I mean is not I mean not a big over it with respect to the computational gain you may buy execution in parallel. If that is so, then you would like to do this operation right.

So, this is some possible mapping that you can make and depending on the kernel compute characteristics it may have some advantage right. So, in general we understand that that scheduling is a complex problem, because you have a architecture with multiple kinds of devices each kernel may have a map mean in some specific way to some device like kernel K 1 may is some specific execution time in a CPU it may give some different execution time on a GPU right.

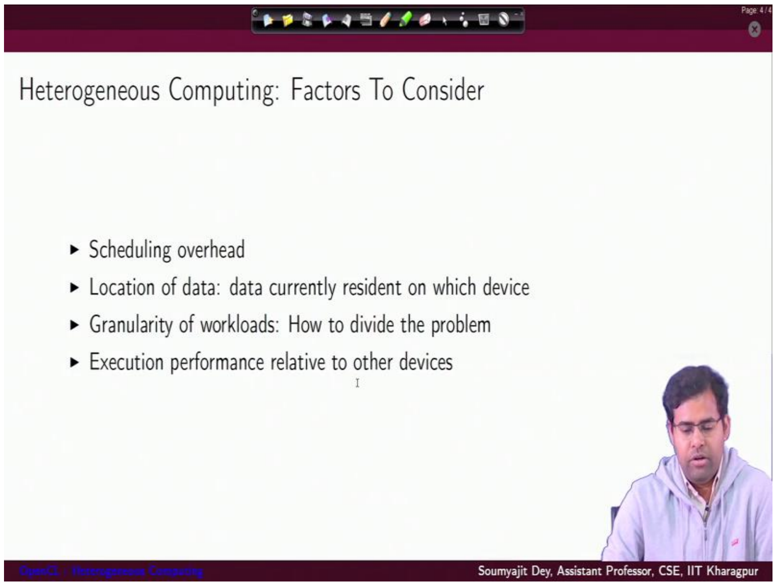
So, first as a programmer you need to identify what is a good device will do the mapping and then you should like to set up the command queues and launch the kernels respecting the dependencies. For example, here as we discussed K 1 and K 2 do not have any dependency, but they are the execution of K 3 depends on K 1 and K 2, both of them producing their outputs. So, K 3 I mean, I need to synchronize before executing K 3 and then if required, if it is advantageous, then I may want to partition K 3 and execute.

So, as we can see that there can be many possible options like it will also make sense, let us say it all depends on the runtime scenario that if K 2 is producing something and K 2 is mapped to a device shall I map gets you to the same device, well, when we lay want to do that, so, I will want to map K2 and K3 has to the same device in a situation because then this data transfer over it from K 2 to K 3 will not exist right.

So, we have to figure out what is a good mapping will I mean in case K 2 is idle or I mean, then I mean, the device work it has been mapped is idle, then that also makes sense right. So, how to choose a mapping, what is the best mapping for a given application and an architecture is something that you have to fix. Here we are not discussing that problem. We are just trying to tell that will multiple mappings can be possible given an application and an article lecture as a programmer; you need to figure out what are the mapping characteristics.

What is a benefit, in which case and accordingly decide. So we are just trying to motivate that, if I map K 2 and K 3 together in a single device, it may be good. Because then I do not have this data over head for maybe when I mean there is something else, maybe some other job is there, who for which I will like to use this device. So then I will actually like to use K 3 to I mean, and map it to some other device, right. So it is all a situation which you have to fix based on the options and all that. So this was one example of that scheduling that we went through.

(Refer Slide Time: 07:47)



Heterogeneous Computing: Factors To Consider

- Scheduling overhead
- Location of data: data currently resident on which device
- Granularity of workloads: How to divide the problem
- Execution performance relative to other devices

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

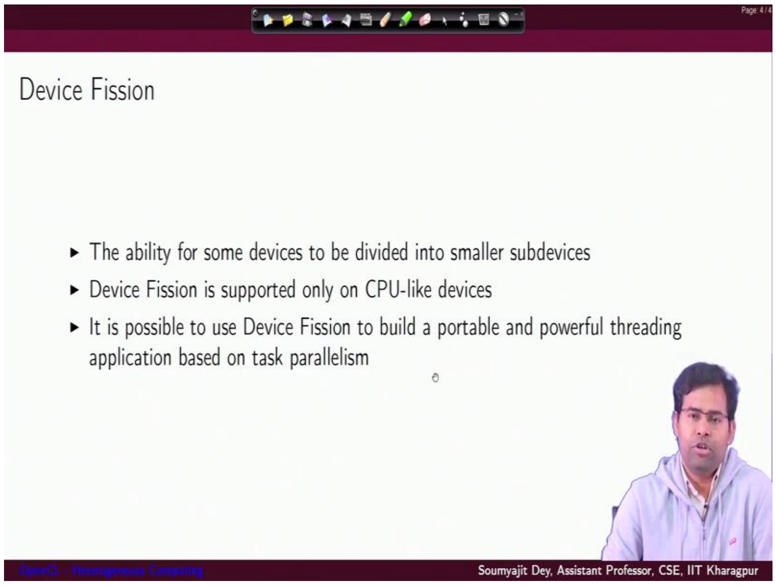
Now so overall, these are the factors that we like to consider that what is the scheduling overhead, right. I mean, if we are scheduling multiple kernels in multiple devices, then we have to do some bookkeeping like we have to wait on events, we have to perform data transfers. So these are the overheads we have to take care of and we have to say that will we will perform a scheduling decision only when the overhead, actually I mean, the benefits outweigh the overheads, right.

So, the other issue would also be that the granularity of workloads that how much you will like to divide the problem, if you divide the problem into too many instances, then for each problem, the number of threads become too thin to sustain any overhead that is also added to the kernel launch event. So, this is important that whenever you are launching a kernel that has some overhead with respect to the runtime

Whenever you are actually doing a notify callback, or you are executing a kernel based on the execute code inclusion status of something else that also has a context switch over it. So, all the switching overheads are there. So if you define divide the problem into too many instances and try to solve it then the overheads will keeping and we I mean far outweigh the advantage of federalism that you may have, right.

So these things one has to figure out. And also you have to take care of whether the execution or what is the execution performance of the different devices. So, it is a heterogeneous compute platform, you may have multiple different GPUs. So a colonel executing in one GPU with a performance metric in a different GPU may have a different performance metric right. So these are all 4 factors that want us to figure out.

(Refer Slide Time: 09:31)



Device Fission

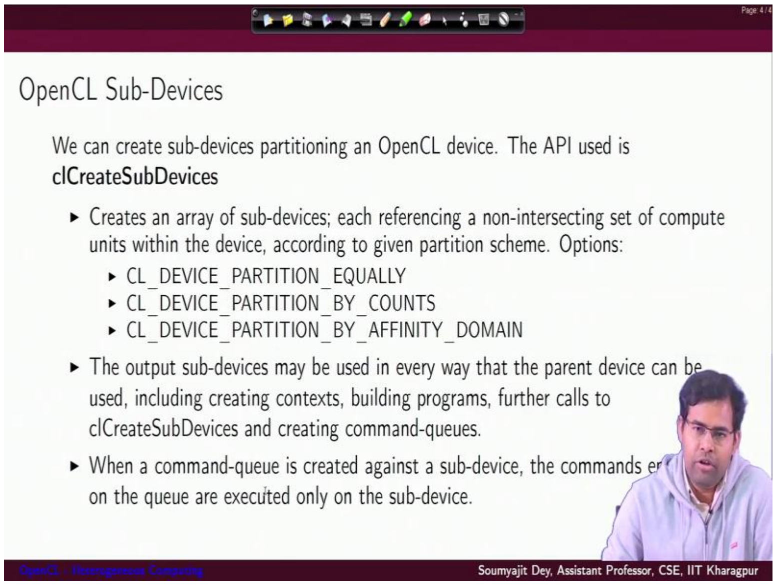
- The ability for some devices to be divided into smaller subdevices
- Device Fission is supported only on CPU-like devices
- It is possible to use Device Fission to build a portable and powerful threading application based on task parallelism

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

The other topic we like to cover is device fission like there are several CPU devices from different vendors, which support this concept of dividing the device in the OpenCL runtime system into smaller sub devices. For example, in general, if you have one CPU, OpenCL will identify it as a device, right but an enzyme discipline, I mean, as CPU a model CPU with multiple assembly units, it can be logical partition to several sub devices. In that case OpenCL system can identify them as separate sub devices and manage them separately, right.

So, till that this idea of device fission is supported only on CPU like devices. And it is possible to use device vision to build a portable and powerful threading application based on task parallelism. We will like to think that if there are more such sub devices, then they can actually perform that overlapped execution to provide a more to explore task level parallelism and give us some optimized execution of our given assembly kernel.

(Refer Slide Time: 10:41)


OpenCL Sub-Devices

We can create sub-devices partitioning an OpenCL device. The API used is **clCreateSubDevices**

- Creates an array of sub-devices; each referencing a non-intersecting set of compute units within the device, according to given partition scheme. Options:
  - CL_DEVICE_PARTITION_EQUALLY
  - CL_DEVICE_PARTITION_BY_COUNTS
  - CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN
- The output sub-devices may be used in every way that the parent device can be used, including creating contexts, building programs, further calls to clCreateSubDevices and creating command-queues.
- When a command-queue is created against a sub-device, the commands en on the queue are executed only on the sub-device.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

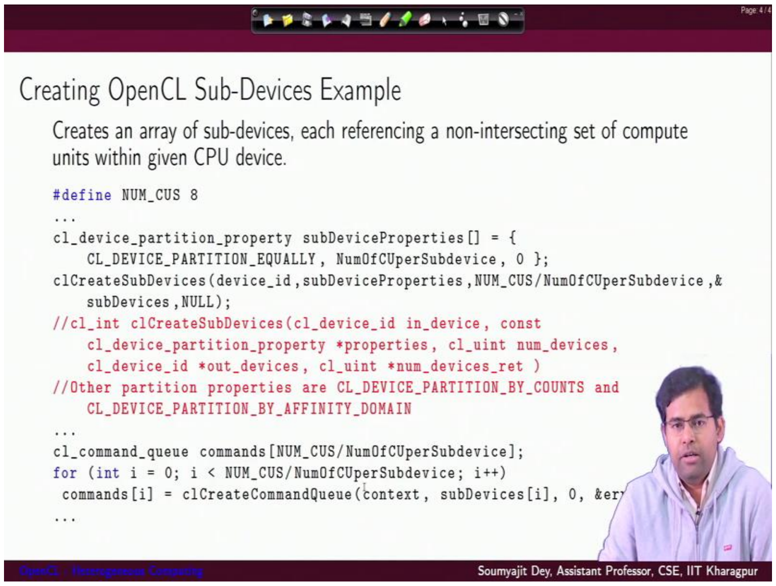
So will not go in great detail about I mean, executing a program in multiple sub devices, but rather let us just discuss how to create such as the prizes. So these are the options that are available till DAG in OpenCL runtime system, you can use this API functions to create sub devices, but how you want to create the device partition, there are several options. So, you can create an array of sub devices using this call, each of the sub device will refer non intersecting set of compute units.

So, suppose you have a set of compute units in each sub device, you will have a set of compute units which will not belong to the set of computers for some other device. And so, this partitioning of devices the internal SM the units of the CPU across sub devices can be has to be managed and you can give as a programmer you can give different directives that how to manage them. So, you can tell the runtime system that you will you use if you use this flag CL DEVICE PARTITON EQUALLY and then it will perform the allocation of sub i mean computers to sub devices following this flag.

You can give it a by counts flag and also there is this option of affinity domain right. So, there are different possibilities in which you can actually partition the sub device space. Once the sub devices are created, they can be they are very much like normal OpenCL device. For them, you can create contexts, you can build programs you can launch programs into command queue specified for sub devices and all that, right.

So, these options you can just look into, we will just give a program example with 1 of the options, but of course, these are quite intuitive. So and for a specific sub device, you can create a separate command queue and you if you launch a command, it will get launched not into the entire device but into that sub device.

(Refer Slide Time: 12:47)



Page 4 / 4

Creating OpenCL Sub-Devices Example

Creates an array of sub-devices, each referencing a non-intersecting set of compute units within given CPU device.

```
#define NUM_CUS 8
...
cl_device_partition_property subDeviceProperties[] = {
    CL_DEVICE_PARTITION_EQUALLY, NumOfCUperSubdevice, 0 };
clCreateSubDevices(device_id,subDeviceProperties,NUM_CUS/NumOfCUperSubdevice,&
    subDevices,NULL);
//cl_int clCreateSubDevices(cl_device_id in_device, const
    cl_device_partition_property *properties, cl_uint num_devices,
    cl_device_id *out_devices, cl_uint *num_devices_ret )
//Other partition properties are CL_DEVICE_PARTITION_BY_COUNTS and
    CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN
...
cl_command_queue commands[NUM_CUS/NumOfCUperSubdevice];
for (int i = 0; i < NUM_CUS/NumOfCUperSubdevice; i++)
 commands[i] = clCreateCommandQueue(context, subDevices[i], 0, &er
...
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So let us have a look into this simple program. So, first thing, you have to set up a device property array using which you are trying to specify how the sub devices are defined right. So, for this you have to launch you have to define an array let us say we give it the name sub device properties is type has to be CL device partition property, and you have to give the up to populate this area with the following elements. The first component has to be the primitive like what is your choice of partitioning scheme.

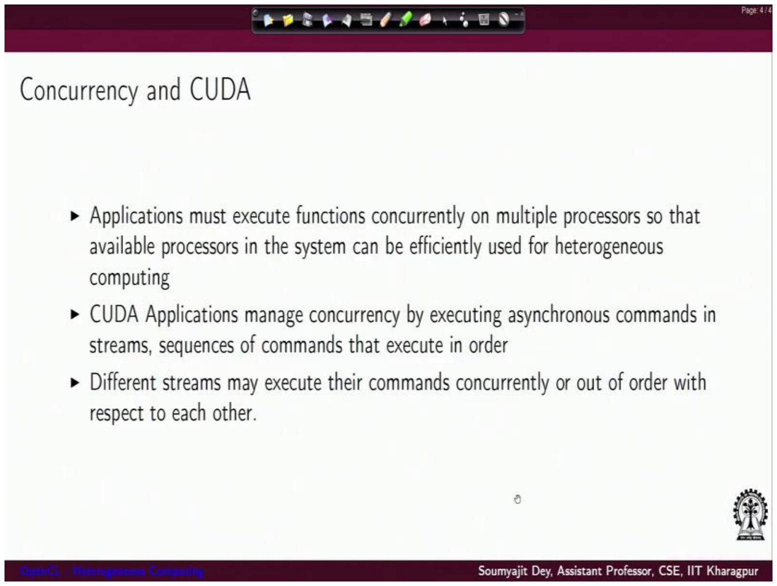
So let us say I choose CL device partition equally distribute the sub devices equally and you have to specify; what is the number of sub devices you want. So obviously, the first thing is CL device partition equally. So whatever is the number of units available, you should do at saying that will you divide them equally across the number of sub devices and the number of sub devices you want is specified by this flag.

So once this array set up, you just make a call CL creates sub devices, you give it the device ID and you pass your requirements to this array and you give this thing that is what is the total number of compute units you have and divided by the number of compute units to one per sub device and this call will actually provide back the handles to each of the sub devices inside this array sub devices.

So, once this is done next you let us look into how you can actually create queues and engage the different sub devices. So, and also we like to say that here we are using the equal equally flag instead of that I can actually give explicit counts or I can specify an affinity domain right. So, coming back to the command queues. So this is the array in which I am trying to define command queues for each of the devices.

So in the fall loop as you can see, we are firing the CL create command queue command and the number of times this loop is going to rotate is exactly equal to the number of sub devices, which is the total number of compute device a compute units divided by number of compute units 1 per sub device. So that is the number of sub devices you have created. And for each of them, you are creating a separate command queue right.

(Refer Slide Time: 15:23)



Concurrency and CUDA

- Applications must execute functions concurrently on multiple processors so that available processors in the system can be efficiently used for heterogeneous computing
- CUDA Applications manage concurrency by executing asynchronous commands in streams, sequences of commands that execute in order
- Different streams may execute their commands concurrently or out of order with respect to each other.

OpenCL - Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now so, that is how we actually create the set of sub devices and you create separate command us for each of the sub devices. Now, what is the advantage of this like we have been discussing?

Well, now you can issue commands in parallel across the different command queues for the sub devices, and that will help you to achieve overlapped execution in case you want 1 sub devices to do something you want some other subdivides to do some other thing.

Let us say you want sub devices 1 to some data copy operation while sub device 2 you actually engaging to some executing some other kernel. So, like once the sub devices are created, they are really going to act like normal full-fledged OpenCL devices to the programmer is point of view right. So with this discussion about device fission and properties of sub devices, we like to end this lecture. Thank you.