

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

**Lecture # 52**  
**OpenCL – Heterogeneous Computing (Contd.)**

Hi, welcome back to the lecture series on GPU architectures and programming. So, in the last lecture we have discussed how to create open CL sub devices how to perform DAC scheduling on architecture comprising multiple devices which support open CL and all that now, in this context while we have been discussing how I can execute kernels in parallel, or how I can break down a kernel and partition it and launch into multiple devices.


We like to go back to our original programming language which was CUDA and check whether in CUDA we have any support for such concurrent execution.

**(Refer Slide Time: 01:05)**

Page 4/4

### 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

If you remember, so, by the way, this is the point where we have we can officially say that we are done with our, discussions on open CL and heterogeneous computing using open CL. We are kind of going back to CUDA programming language and use usage for this purpose and also in the future lectures will be using CUDA for other purposes as I have been provided in the indexes. So, the issue is, if you look at our earlier lectures on CUDA programs.

CUDA runtime system and everything, our programming model has been used to launch a host code, the host code execute sequentially and it can launch a CUDA kernel and then it will wait till the kernel returns and then you can fire and the host code can go beyond and then it can launch another CUDA command. So although the host code can launch the CUDA kernel and go beyond to the next command to be executed for the GPU.

But the next command it will mean execute only when the kernels corners execution is done. So there is sequentiality in that way. But can I have a CUDA applications sharing, I mean, a GPU and walking together, because as we see, the issue is, if I look at a GPU is like an accelerator, it does not have any scheduling primitive from the software side is a hardware scheduler where is device. So, you cannot launch I mean, the way we have defined till now, we never had the facility that we will be launching multiple kernels together on a CPU on a GPU device at the same time to execute or something else.

So the issue is, can I have support for concurrent execution CUDA? The answer is yes. In currently available CUDA runtime systems, you have such supports. So and that is also desirable because in then you can also extract the maximum parallelism that you have ever in the underlying hardware. So, the way CUDA manages concurrency is by executing a synchronous commands in streams or CUDA streams and in each stream can execute a sequence of commands that are that have been that have been dispatched to the stream in that order.

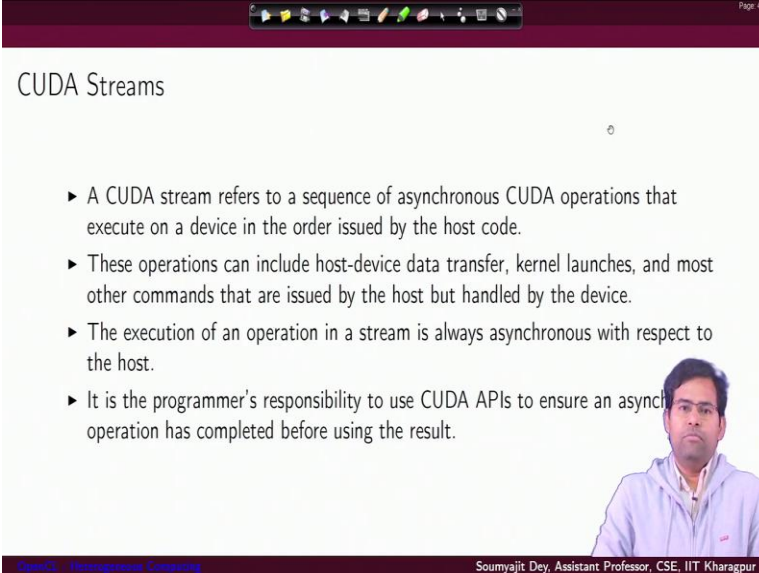
So in CUDA the way we have been discussing till now we have when we are talking about a CUDA program, there is something called a default CUDA stream that stream is 0. In general, I can write a program which is engaging this CUDA, GPU by running multiple CUDA streams in parallel. So the runtime system will use this obstruction of streams to execute CUDA commands in parallel, while each of the streams while the host program will just issue commands as synchronously to each stream.

So we will see that why we call it synchronize and all that so, different streams where I mean, we are so, the host program as we are saying, instead of writing a normal CUDA program, which is a single stream of execution, the host program can be find multiple streams. And it can be it can

dispatch commands for the runtime system individually to each of the streams. And each of the streams will be executing those read/write or kernel commands.

In the order they have been dispatched, but in what order they do it among streams is completely I mean defined by the runtime system. So they will actually execute out of order with respect to each other that is at the stream level.

**(Refer Slide Time: 04:46)**



The slide is titled "CUDA Streams" and contains the following text:

- ▶ A CUDA stream refers to a sequence of asynchronous CUDA operations that execute on a device in the order issued by the host code.
- ▶ These operations can include host-device data transfer, kernel launches, and most other commands that are issued by the host but handled by the device.
- ▶ The execution of an operation in a stream is always asynchronous with respect to the host.
- ▶ It is the programmer's responsibility to use CUDA APIs to ensure an asynchronous operation has completed before using the result.

In the bottom right corner of the slide, there is a small video inset showing a man with dark hair and a beard, wearing a light blue hoodie, looking towards the camera.

At the bottom of the slide, there is a footer with the text "OpenCL: Heterogeneous Computing" on the left and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur" on the right.

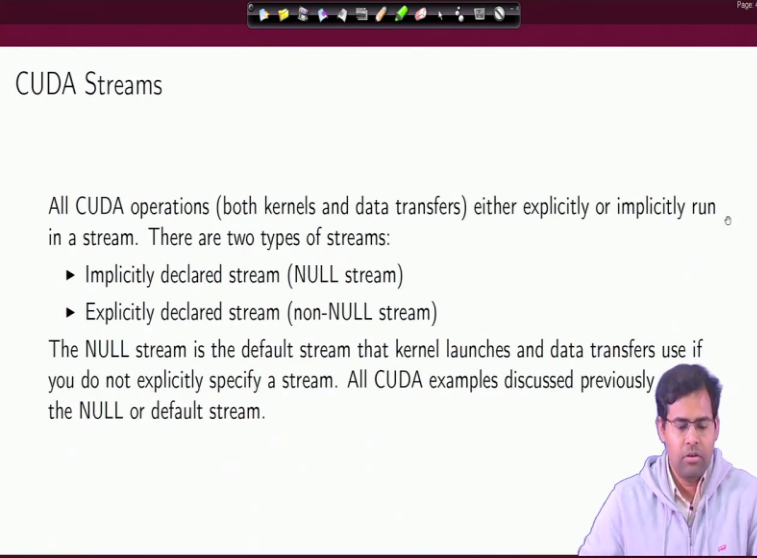
So what is a CUDA stream? So it refers to a sequence of synchronous CUDA operations that execute in a device in the order issued by the host code. So suppose the host code is issuing a set of orders to put a stream 1 those will be executed by that stream in that order. The host code is I mean also issues some other commands to a separate stream that stream will execute whatever commands has been issued to that stream in that order.

But as I was saying, in what order these streams interleave is dependent on the runtime and there is no finite control on that so, the typically when I talk about CUDA commands, we are thinking of basic operations like host to device your transfer host and device or device to host your transfers, kernel launches and synchronization commands that may be issued by the host but they are handled by the device, those kinds of commands, and the execution of operations inside a stream that is synchronous with respect to the host.

Now, this is the key point. So the host program just dispatches commands, 2 different streams and moves forward. So this commands execute in a CUDA stream setup for the GPU device completely independent of what the host is doing the host has dispatched commands is very much like open CL command queue, you have open CL host program moving forward dispatching commands to command queues, you can have the CUDA program moving forward and dispatching operations to the stream.

So, I mean, once they have been dispatched, the host program does not have a control, it is just moving forward as synchronously. So, now it becomes the job of the programmers to actually use the CUDA APIs to ensure that these asynchronous operations are can preserve dependencies. So here also like open CL, we have been discussing here also will see the role of events through which synchronization can be attained across streams and all that.

**(Refer Slide Time: 06:47)**



The slide is titled "CUDA Streams" and contains the following text:

All CUDA operations (both kernels and data transfers) either explicitly or implicitly run in a stream. There are two types of streams:

- ▶ Implicitly declared stream (NULL stream)
- ▶ Explicitly declared stream (non-NULL stream)

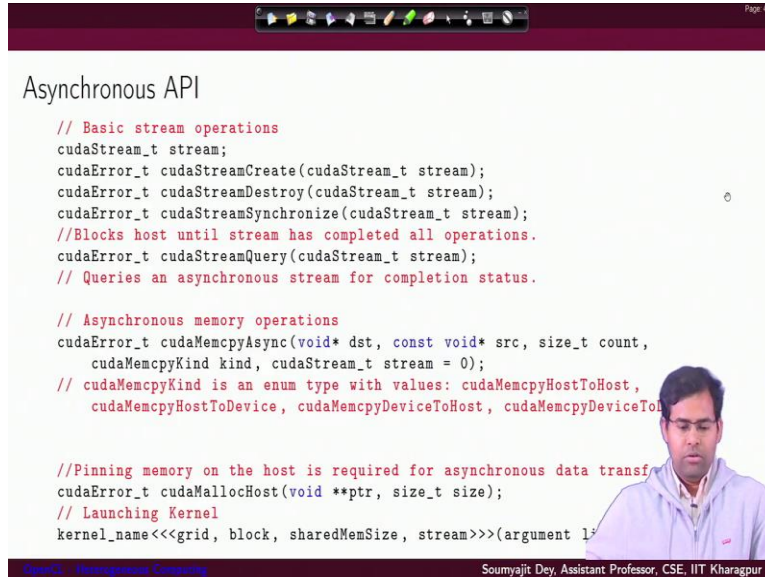
The NULL stream is the default stream that kernel launches and data transfers use if you do not explicitly specify a stream. All CUDA examples discussed previously the NULL or default stream.

The slide also features a video inset of a man in a light blue hoodie in the bottom right corner. At the top right of the slide, it says "Page 6/4". At the bottom left, it says "OpenCL - Heterogeneous Computing" and at the bottom right, it says "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

So by speaking of CUDA streams, all CUDA operations, both kernels and data transfers, they either implicitly or explicitly running a stream. So when we do not specify the stream is that default, now stream. So these are the 2 defined 2 different types of streams the implicitly declared stream or the NULL stream. And the explicitly declared streams, which are the non-NULL stream. So, by default everything is stream 0. And otherwise, if you want to create more streams for attaining concurrent execution, you have to define them exclusively.

So the NULL stream is the default stream where kernels launch and data transfers, use in specify it and whatever, like we have been saying that whatever good examples we specified earlier, they were all by default getting launched into the string.

**(Refer Slide Time: 07:36)**



```
Asynchronous API

// Basic stream operations
cudaStream_t stream;
cudaError_t cudaStreamCreate(cudaStream_t stream);
cudaError_t cudaStreamDestroy(cudaStream_t stream);
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
//Blocks host until stream has completed all operations.
cudaError_t cudaStreamQuery(cudaStream_t stream);
// Queries an asynchronous stream for completion status.

// Asynchronous memory operations
cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count,
    cudaMemcpyKind kind, cudaStream_t stream = 0);
// cudaMemcpyKind is an enum type with values: cudaMemcpyHostToHost,
    cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice

//Pinning memory on the host is required for asynchronous data transfer
cudaError_t cudaMallocHost(void **ptr, size_t size);
// Launching Kernel
kernel_name<<<grid, block, sharedMemSize, stream>>>(argument 1)
```

So, let us get our first test of this asynchronous API. So, first we are just going to give it I mean, we are just trying to give some basic examples of stream operations like so the type is CUDA stream underscore t. So you can define a stream you can declare a stream using this command, the first one who CUDA stream underscore t then stream. So this is a stream that you have defined. And then after declaring it you have to create the stream using a command like CUDA stream create.


If you want to destroy an already created stream, you have to fire the command CUDA stream destroyed. And if you want a stream to synchronize, and we will see that first is required, you have to use this command CUDA stream synchronize. Now so when you want to synchronize, you actually blocked the host until all the streams has completed operations. So we will see the different possible synchronizations. Maybe this is a good time to look at that, like just a minute.

**(Refer Slide Time: 08:46)**

Page 4/4

## Synchronization

- ▶ Synchronizing the device `cudaDeviceSynchronize()` : wait until all streams finish
- ▶ Synchronizing a stream `cudaStreamSynchronize(stream)`
- ▶ Synchronizing an event in a stream `cudaStreamWaitEvent(stream, event)`
- ▶ Synchronizing across streams using an event `cudaEventSynchronize(event)`



OpenCL - Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So these are the different ways you have synchronization on the stream. Now we are just saying that let us say we just had this command `CUDA stream synchronize`, then you are asking that wait if this is they are in the host program in the program. We will wait here until all the streams finish. If this is there in the host program `CUDA streams`, sorry, the first one is `CUDA device synchronous`. So if you are firing `CUDA device synchronize` in the host program.

You want all that streams that are running in the `CUDA device` to finish, and then only the host program move beyond this point. If you want a specific stream to synchronize, then you just write `CUDA streams synchronize with the stream` that means the host program does not move forward until this specific stream will finish the other ones will see soon. So now, let us go back to our example that we have started. So this is `CUDA streams synchronize`.

So you have blocked the host program until unless the stream finishes, whatever operation it has been given here, we have not given to anything till now. So the other thing that you can do is you can query a stream. So, I mean, this gives you an error status. I mean, some status can say whether it is completed either executions were there, it is still in process and all that so, an important thing that we have to remember is when you want to perform data transfer to the `CUDA streams`.

They have to be a synchronous. Why? Because unlike a normal CUDA program, if you are using CUDA streams, then your host program is moving forward just by dispatching commands. So for that the memory copy operation should be done in such a way that it is complete is it has to run in a separate thread with respect to synchronously with respect to the real host programs, thread of execution.

So for that, unlike normal CUDA mem copy, you have to use this mem copy command which is CUDA mem copy async. Because now you really want a synchronous memory operation that you do not wait at this point here until the mem copy is done. You just issue the command mem copy to a specific stream. Let us say in this case you stream is 0 and you just move forward. So there is a difference between a normal CUDA mem copy and mem copy assigned.

So, this means that transfer a synchronous so that the host program can move at its own speed and this transfer can happen at own speed as is supported by the device. Now in this command as you can see where you have a source, you have a destination you specify in which stream to copy, you give the size and you also have a flag CUDA mem copy the kind, which is an enumerated type. It has the possible values like we all know that earlier. I mean these are also similar to the flags we have been using for CUDA mem copy normal mem copy.

So you can either do mem copy host to host does not make sense maybe and then you can do host to device, device to host and if you want to do mem copy across multiple devices, then device to device. Now there is another important thing when you are doing this I mean work on across streams. And then the requirement would be that you want whatever memory has been allocated on the host side to be pinned. So let us understand what we say here like.

Let us say you have a host side memory and a buffer has been declared and you want to copy it to a device side memory. So, this is the host memory buffer and you were copying it to a device. Earlier it used to be a single 1 short copy command. But now, in the current form, since things are a synchronous, so the copy you are actually living it to the copying us from the device and the runtime system to manage a synchronously so the copy now can happen as and when the runtime system wants to do it.

While you were you are moving forward with the host program and the copy operations are happening as synchronously. The point is they may not happen in 1 shot, it is not a blocking operation. So, then thinking from the operating systems point of view, it may happen that the copy operation started some amount of data has been copied, and then when you want to restart by that time, the page in those squares I mean host side is memory, where this buffer was sitting, it has been mapped out.

So, that will create unnecessary delay and it will be an issue. So, what you want is if you are allocating memory on the host side, this is the host side memory from which you are going to copy in synchronous streams. You want this memory to be what we call us pinned. You want this memory to pinned on the device so that you are essentially we are asking that, you do not pay it out because I am going to copy as and when I want in a lazy fashion.

So for ensuring that the memory is pinned on the host side, unlike normal CUDA malloc, now you use the command CUDA malloc host. So, there are differences you can see CUDA mem copy async, CUDA malloc host. So, you use CUDA malloc host to have a pinned data offering the host side memory, and then you launch the kernel to a specific stream. So, this is a new thing earlier they have been launching kernels. So, you give a grid and block then as you can see, there can be a third parameter that will how much amount of shared memory in the GPUs will be allocated for this specific kernel.

That can also be specified by this shared memory size. Why will you are having multiple streams it is expected you can launch multiple kernels across streams. So you may want to specify well which stream working with which kernel should use how much amount of shared memory, so that is where you can specify the shared memory size result for this kernel. And in which stream, do you want the kernel to be launched?

**(Refer Slide Time: 15:18)**



CUDA Streams: Basic Example

```
__global__ void kernel(float *g_data, float value)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    g_data[idx] = g_data[idx] + value;
}
#define CHECK(call)
{
    const cudaError_t error = call;
    if (error != cudaSuccess)
    {
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);
        fprintf(stderr, "code: %d, reason: %s\n", error,
            cudaGetErrorString(error));
    }
}
```

OpenCL: Programming Concepts Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So this is how you can actually copy, I mean, you could declare host side data, data buffer, and then you can copy that so this is not just a little reminder, this is not a sequence of program statements. We have just written these different directives to let you understand what are the different specific operations that the synchronous API of CUDA is actually allowing you to do? So here is how you launch the kernel.

These are you launch the kernel to the host in a pinned fashion these are you perform a synchronous transfer. These are you actually synchronized on a specific stream and is how you create or destroy the stream. And you can also query a stream status. Coming to basic CUDA stream programming examples. So let us say you have a normal kernel. So all you are doing is you calculate the global thread id. And you just add that with a constant value, the simple kernel.

So what we will do is we will also define a check function here, so that whatever function call will make next in whatever programs we show you, the status that the function called returns can be, is actually handled nicely through this check function. So I mean, we are actually doing this so that we can avoid writing all these handler code everywhere. So for every function call, we can just give this call to check so that whatever type will be emitted. If in case it is not a success, it is gracefully handled with suitable file prints to their era standard.

**(Refer Slide Time: 16:56)**

Page 5/5

## CUDA Streams: Basic Example

```

int main(int argc, char *argv[])
{
    int devID = 0;
    cudaDeviceProp deviceProps;
    CHECK(cudaGetDeviceProperties(&deviceProps, devID));
    printf("> %s running on", argv[0]);
    printf(" CUDA device [%s]\n", deviceProps.name);
    int num = 1 << 24;
    int nbytes = num * sizeof(int);
    float value = 10.0f;
    // allocate host memory
    float *h_a = 0;
    CHECK(cudaMallocHost((void **)&h_a, nbytes));
    memset(h_a, 0, nbytes);
    // allocate device memory
    float *d_a = 0;
    CHECK(cudaMalloc((void **)&d_a, nbytes));
    CHECK(cudaMemset(d_a, 255, nbytes));
}

```

OpenCL - Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So now let us look into the host programs example for CUDA streams. So, inside your main, you first declared this device properties, you have a variable for device properties. And you can check whether I mean for this device properties for some specific CUDA device id, what are the device properties, and then it will be able to say that will what is running in that CUDA device and all that.

**(Refer Slide Time: 17:35)**

Page 5/5

## Concurrency Example

```

> ./simpleMultiAddBreadth Starting...
> Using Device 0: Tesla K40m
> Compute Capability 3.5 hardware with 15 multi-processors
> CUDA_DEVICE_MAX_CONNECTIONS = 1
> with streams = 4
> vector size = 262144
> grid (2048, 1) block (128, 1)

Measured timings (throughput):
Memcpy host to device : 0.383424 ms (2.734769 GB/s)
Memcpy device to host : 0.182272 ms (5.752809 GB/s)
Kernel : 1605.997192 ms (0.001306 GB/s)
Total : 1606.562866 ms (0.001305 GB/s)

Actual results from overlapped data transfers:
overlap with 4 streams : 402.014435 ms (0.005217 GB/s)
speedup : 74.976738

```

OpenCL - Heterogeneous Computing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, if you look at an example of the prints we have here, so, is just going to say that will, and this is the program that we are running, and it is running in that this specific device with device capabilities and all that so, I believe, earlier we have already talked, about what our device properties I mean, what are the different CUDA device properties and all that essentially the

information of the runtime system So, once I make this call using this, I am actually extracting from the runtime system all the device properties and storing them into this.

And for the specific for some specific GPU device id in this case we are said they were just waiting for the ideal GPU device with id is 0, it can be whatever you want? So and then we have some standard prints just to tell what is the program name that is running because it is again, that is that is what should be the executable name in argv is 0 location. And you also print what is the name of the CUDA device what is the GPU device on which you are running the program.

So that would be in the device properties structure itself in the field device properties. So these just normal bookkeeping code like we write for you have the device property structure, when you define an order type for that you define a device property structure and you extract the device properties for the target device, you print all those status effects. And then let us get into the normal action. So, let us say you start with elevating host side memory using as we have discussed now, we will be using cooler CUDA malloc host command to get in host memory.

And then you can use normal CUDA malloc. For the device and memory why use normal CUDA malloc? Well, then in the in the device side memory, you do not have an issue with paging out of memory and all that so, as you can see that we are making all those function calls and under they got of the check statements, so that any kind of error is was fully handled. So once this is done, let us decide on some launch configuration of the kernel. So, you define a set block of size 512, you define a grid with this some specific number of blocks like this.

**(Refer Slide Time: 20:15)**


Page 5/5

## CUDA Streams: Basic Example

```

// set kernel launch configuration
dim3 block = dim3(512);
dim3 grid = dim3((num + block.x - 1) / block.x);
// create cuda event handles
cudaEvent_t stop;
CHECK(cudaEventCreate(&stop));
// asynchronously issue work to the GPU (all to stream 0)
CHECK(cudaMemcpyAsync(d_a, h_a, nbytes, cudaMemcpyHostToDevice));
kernel<<<grid, block>>>(d_a, value);
CHECK(cudaMemcpyAsync(h_a, d_a, nbytes, cudaMemcpyDeviceToHost));
CHECK(cudaEventRecord(stop));
// have CPU do some work while waiting for stage 1 to finish
unsigned long int counter = 0;
while (cudaEventQuery(stop) == cudaErrorNotReady) {
    counter++;
}

```



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

So this is my definition of the grid? I mean, what is the number of blocks and all that so once I have defined what is the total number of blocks? That would be in the grid, and what is the thread block size? Then let us use the CUDA API event to perform some bookkeeping of the execution times. So I believe in our earlier lectures, we have already discussed of the CUDA event API and how to use CUDA events to sample timings.

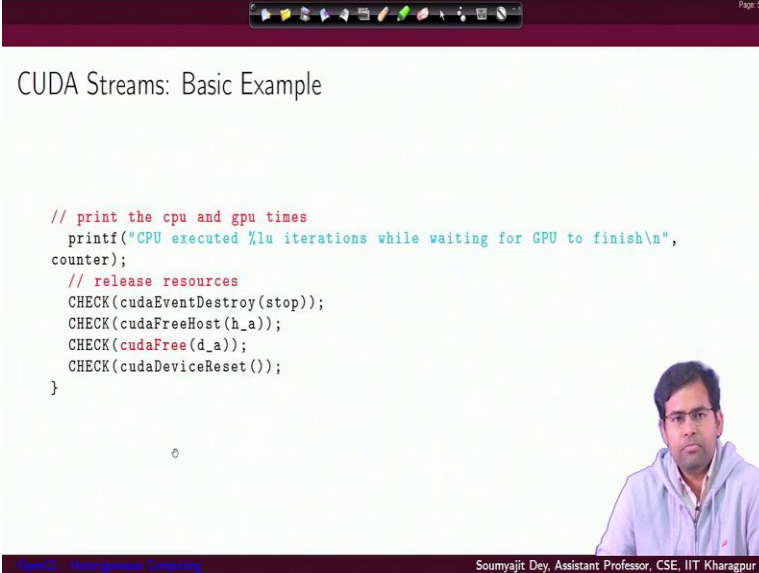
So at this point, you use the CUDA event create command to sample the current time in and storage in the event, object type event create and stop. And then let us say we get into the action with the different CUDA streams. So we will just issue as work as synchronously to the GPU, which is said to me, do I mean GPU stream 0. So what we are doing is we are just using the CUDA mem copy as in commands to launch a copy of version from the host side array hA to the device side array.

Using CUDA mem copy host to device directive. And as you can see, that is an asynchronous copy. So we will just immediately go forward to launching the kernel. And since no stream is explicitly specified is always happening in stream 0. And then once that is done, you expect the result to be in the in the device side memory. So you get back that data from the device side to the host side memory, and then you again record you perform CUDA event record again on the stop event.

So, like, so, here, you create you actually declared the CUDA event object stop and you actually created you use this API function could even create to define and then here you actually sample it so, from this point, the event is ready and it has started sampling the timings. And you are actually measuring the time that has been sampled here in using this CUDA event record. Now after this, you want the CPU to do some work while waiting for stage one to finish that you are you are waiting for the stream to complete.

So what do we is you can keep on this job. And whether it is at a state that is, it is not ready, whether it is returning or not ready flag. And for the time being, you are just making the CPU some do some simple action like, like implementing a variable, because as you can see that all the work of the memory copy is going on a synchronously to the GPU device and coming back is just there to the host side here.

**(Refer Slide Time: 23:28)**



The screenshot shows a presentation slide with the title "CUDA Streams: Basic Example". The slide contains C++ code for resource cleanup. A video inset in the bottom right corner shows a man in a light blue hoodie. The slide footer includes "Page 5/10", "Parallel Programming", and "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

```
// print the cpu and gpu times
printf("CPU executed %lu iterations while waiting for GPU to finish\n",
counter);
// release resources
CHECK(cudaEventDestroy(stop));
CHECK(cudaFreeHost(h_a));
CHECK(cudaFree(d_a));
CHECK(cudaDeviceReset());
}
```

So once we know that, well, from this event record, we get to know that, things are done, you move forward. So from the CUDA event enquiry, you get the stream status, that, these commands have finished their execution, then you actually print the CPU and GPU times. So you get out of this loop when you are actually querying this event and you are getting that the good I mean, it is not really good era not ready for this, this flag is no more true.

Then you just print what is the execution time of the CPU? I mean, what is the number of iterations that the CPU spent waiting for the GPU to finish, because that status will actually change when this all the commands that have been synchronously issued to the GPU, they are all completed. So this is us telling you how many iterations of the loop happen for the time in which the GPU got it has synchronously executed the kernel.

And then copied back to the last mem copy from device to side. So once everything is done, you just release the resources through a CUDA event destroy because now this event tag is no more required, the CUDA event variable is no more required. And of course, you can free the host side as well as a device that memory and perform CUDA events device reset. So these are a basic example all we are doing is we are doing the normal kernel operations. But just using a synchronous memory copy and a synchronous execution of the kernel.

**(Refer Slide Time: 25:10)**

Exploiting Concurrency

Asynchronous, stream-based kernel launches and data transfers enable the following types of coarse-grain concurrency:

- ▶ Overlapped host computation and device computation
- ▶ Overlapped host computation and host-device data transfer
- ▶ Overlapped host-device data transfer and device computation
- ▶ Concurrent device computation

Page 3/3

Parallel Programming Concepts

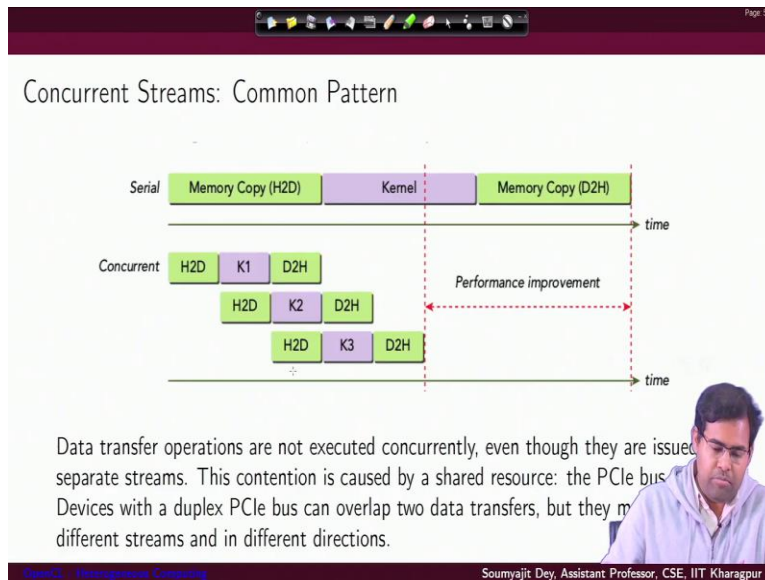
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

But yet, we have not really exploited the concurrency that is available on the GPU side device by using multiple streams, it was just about exploiting this notion of asynchronous dispatch. So now, we will move forward with that and see how using multiple streams, we can have overlapped computation of host and device overlap computation in the host and also a data transfer happening between host and device.

So these are the different possibilities where I can have some action happening both in the host and the device. And in that way, I have overlapped executions with a total execution time reduces? So that is what we want to do. Now this is a listing like what are the things that are possible in parallel host is computing something while and the device also computing something host is computing something and you have some transfer of data from the host to device.

You have similar transfers and also the devices computing something or you have multiple streams that are executing in the device. So, now, the mean these are the possibilities that you have.

**(Refer Slide Time: 26:20)**



Now, the typically common pattern that we have with respect to concurrent execution, I mean the advantage of concurrent execution would be like this for example, consider a normal execution of a GPU program or a CUDA kernel here so, you have a memory copy from host to device that will take some time is followed by the execution time of the kernel. And then you have time for memory copy from that device to the host side.

Now, suppose you break down these memory transfers to small chunks, and you create, I mean different kernels. For kernel versions, I would say for an operating on these chunks. And then you launch them in multiple streams. So, as you can see that these are pipeline execution. So

now, while kernel is executing for this first chunk of data that has been copied in parallel, you can actually have the GPU getting the data for the second instance of the data segment.

So, this small host to device computation is done, then the kernel is working on it in one stream. And in the other stream, you can have the risk the second segment of data chunk coming in from the host side, well in hardware, this should definitely be possible. So you need support like in your GPU side, you have a copy engine, which is copying the data from the host. And you have execution units, which are engaging the different threads and processing the data which have been already copied.

So in terms of hardware, this will be possible. That is why you have you can have this overlapped execution, since you have overlap this executions of copy and computation overall as you can see, you have a performance improvement and this is what we want to attain by launching things in parallel across trims. So, in general, if we assume that we have a PCI Express Bus through which data copy is happening, and of course, we cannot support multiple copies in a in a PCI Express Bus.

Then data transfers cannot execute concurrently. But they although they are issued in separate streams, but I mean the data transfers can overlap with execution of the kernel as we can see in this example, however, there are devices with duplex PCI express buses, where I can have a support for 2 parallel transfers. And in case they are in different streams and in different directions they can have been in parallel.

So, let us say in stream 1, I have a H2D type copy and in stream 2 I have a H2D type copy that is considered absolutely fine, so with this now maybe we will stop here, and we will see some program examples like how this concurrent streams are useful in the next lecture. Thank you.