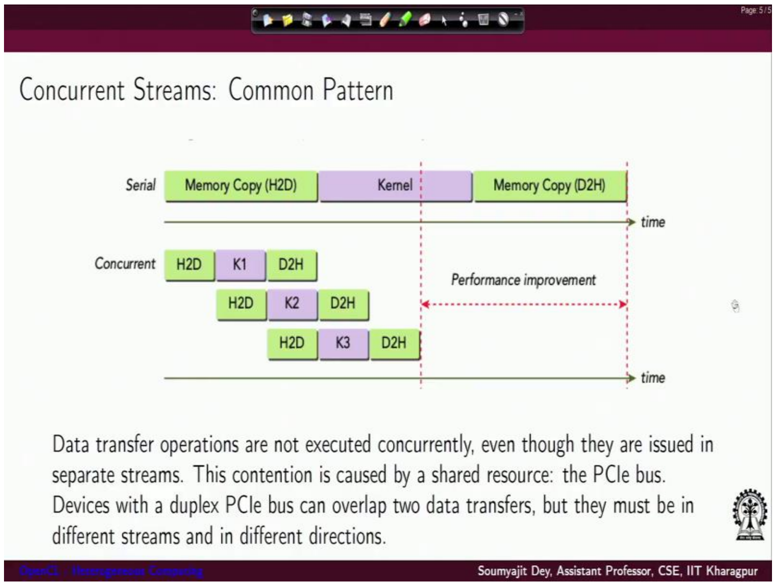
GPU Architectures and Programming

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

Lecture # 53
OpenCL – Heterogeneous Computing (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming. So, in the last lecture we have been discussing about CUDA streams and how overlapped execution can be performed across streams.

(Refer Slide Time: 00:36)



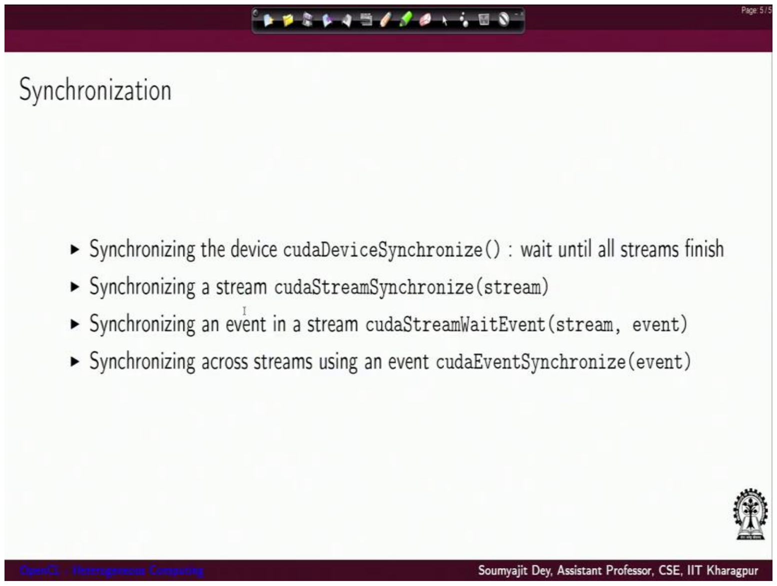
Concurrent Streams: Common Pattern

Data transfer operations are not executed concurrently, even though they are issued in separate streams. This contention is caused by a shared resource: the PCIe bus. Devices with a duplex PCIe bus can overlap two data transfers, but they must be in different streams and in different directions.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And in that regard, we have set up a nice example here of like how we can have overlapped execution in terms of over I mean, I would bet it is better to say overlapping the execution and data transfer. And we also mentioned that in case we have a device where we have a support for duplex PCI express bus then I can overlap today transfers happening in different devices and happening in different directions. So let us assume that we have a normal PCI express bus as well as is the case here so, we can overlap execution in the GPU device along with 1 sided a transfer.

(Refer Slide Time: 01:12)



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)`

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

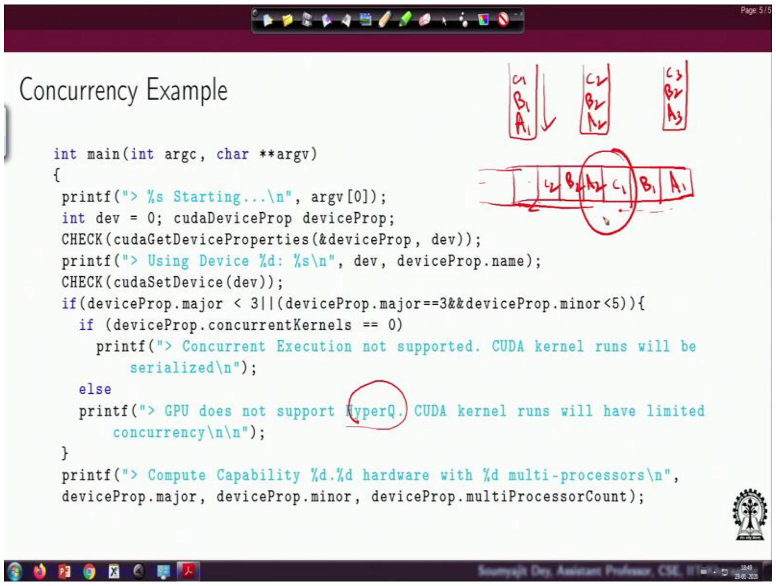
Now, if you are really engaging different streams to perform concurrent execution you may also require sometimes to synchronize the executions because in general streams are asynchronous. So for that CUDA will provide you several synchronization primitives. In case you want the host program to wait till all the CUDA devices all the streams executing on the CUDA device to finish, you can use the first option device synchronize.

If you want a single stream to you if you want the host program to wait a point where you want the single stream to synchronize you execute CUDA streams synchronize with the option that I mean which is the which specific stream you want to synchronize at that point. If you want synchronized an event in a stream on a specific event in a stream, then you have to give this command CUDA stream wait event.

So you are saying that will this stream should wait for this event to happen. So there is again, anyone options, you want all the streams to synchronize at some point. So that is good our device synchronize some at some point in the host problem. If you want to synchronize a specific stream, you use CUDA stream synchronize. If you want CUDA stream wait event, I mean, you want that a stream will synchronize in an in a specific event to a specific event, then you use CUDA stream wait event.

And if you want synchronizing across streams using an event, then of course, you do not need to specify the stream you just specify the event. And you have the command CUDA event synchronize just with the event argument. So you synchronize all the streams with this event.

(Refer Slide Time: 02:58)



Concurrency Example

```
int main(int argc, char **argv)
{
 printf("> %s Starting...\n", argv[0]);
 int dev = 0; cudaDeviceProp deviceProp;
 CHECK(cudaGetDeviceProperties(&deviceProp, dev));
 printf("> Using Device %d: %s\n", dev, deviceProp.name);
 CHECK(cudaSetDevice(dev));
 if(deviceProp.major < 3||(deviceProp.major==3&&deviceProp.minor<5)){
   if (deviceProp.concurrentKernels == 0)
     printf("> Concurrent Execution not supported. CUDA kernel runs will be
         serialized\n");
   else
   printf("> GPU does not support HyperQ. CUDA kernel runs will have limited
       concurrency\n\n");
 }
 printf("> Compute Capability %d.%d hardware with %d multi-processors\n",
 deviceProp.major, deviceProp.minor, deviceProp.multiProcessorCount);
```

So now coming to our concurrency example. So we will try to see how overlap execution can be attained. So let us say you have a main were you in then this part of the code is again, quite simple, you get the device properties, you are saying that in which device what is executing and all that and then you can perform. This is a piece of code that is necessary, I mean to figure out whether you have support for concurrent execution or not. So that is actually I mean, from the device property structure, you can figure out the CUDA runtime systems major and minor number.

And using these numbers, you can figure out how much is the execution support, so in case the measure is less than 3, and or the device measure is equal to 3 and less than 5? Well, in that case, in cases less than 3, you do not have any support for concurrent execution. So CUDA kernels will be serialized now and also there is this issue with major and minor values. So based on this choice, I mean what is the major and minor value of the CUDA runtime system?

There would be I mean, your device property will contain, I mean, whether I mean the value 0 or 1 for the concurrent kernels flag, and that would tell you whether there is support or not. And also, I mean, there are 2 possible checks here, this is what I am trying to say that you can have concurrent execution fully, or you can have limited concurrent execution. I mean, in the other case, which is this else, like you satisfy the major and minor requirements like let us say you have major 3 or minor less than 5.

Then in then in case your CUDA kernels is that mean for those devices, where the major version is earlier to the earlier 2, 3, or it is equal to 3 but the minor version is the minor of number is earlier to 5. For those devices, if you have CUDA kernels command state to 1, for the earlier devices, the thing is, you do not have a support for something called hypercube. So you will have concurrency support, but not fully, we will see what is that and in case you are not getting inside this chip.

That is that means your CUDA runtime systems is saying that will for the GPU device we are talking about the major number is greater than 3 or the major number is equal to 3 but the minor is greater than 5. For that, you have compute capability and which is suitable for concurrent execution. So it will give you your code here, the example we are talking about here, we will just say that, you have concurrent execution capability, and it will also specify what exactly is the major and minor value

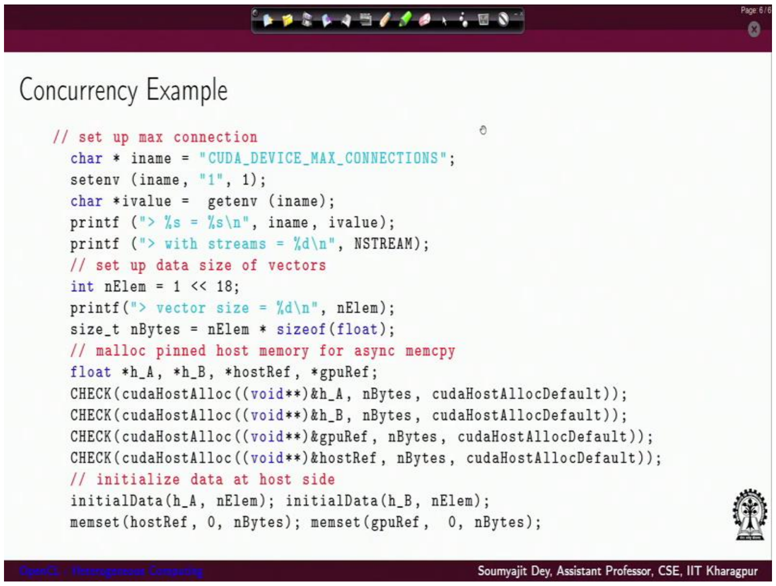
And what is so that would that would be a kind of indicator of what is the amount of compute capability supported. Now, something you have interesting here that what is this Hyper Queue? So, there are earlier GPU devices where you have support for concurrent execution, but there is limited. So, let us try and understand what is this limitation. So, consider the situation that will you have got streams, and you have got multiple kernels, which are executing on the streams.

So you were you want to symbol execute a task graph where you have the kernels A, B and C queued up. Let is call their instances the first second. First all the first instances of A, B and C, you want a to finish then be to work and see what to work on the data that we will be produce things like that maybe you have other instances of these kernels sequenced in some other stream and you have the same corners third instances sequence to some other stream.

The issue is in CUDA older runtime systems, this streams will finally get into a job queue where things are pretty much serialized so you may have I am just trying an option here A1 followed by B1 followed by C1 followed by A2 followed by B2 followed by C2 so on so forth. And then the system is trying to identify what is the parallel is in every level, it does not see any parallelism here because of these dependencies, does not see parallelism here because the dependencies, it only figures that these are the 2 that can be really executed in parallel.

Whereas we can see that will A1, A2 A3 can be executed in parallel B1, B2, B3 can be executed in parallel and C1, C2, C3 can be executed in parallel. So, this issue with me in the final hardware, queue has been resolved with the architectural support of some structure called hypercube, where through which it is actually able to see the dependency structure and see what is the full scope of concurrent execution? That is possible across the different streams. So, we are we are not going to more architectural details, you can read NVIDIA documents that are widely available.

(Refer Slide Time: 08:23)



Concurrency Example

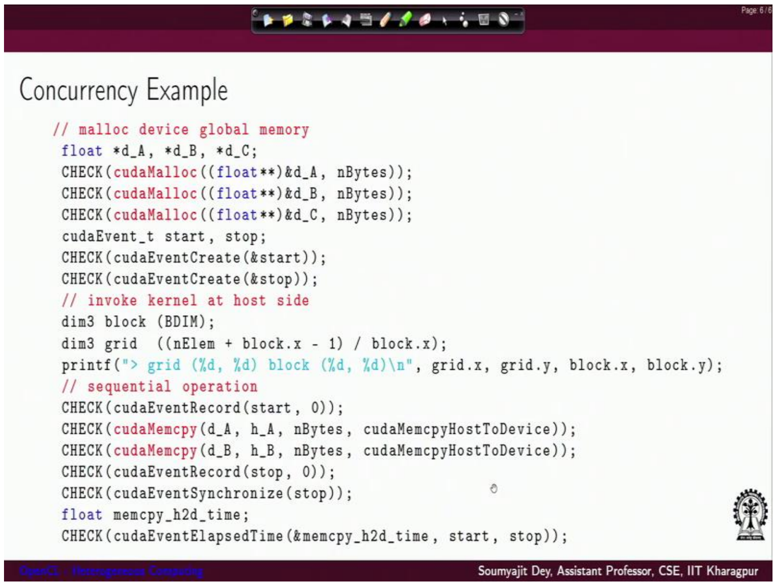
```
// set up max connection
  char * iname = "CUDA_DEVICE_MAX_CONNECTIONS";
  setenv (iname, "1", 1);
  char *ivalue =  getenv (iname);
  printf ("> %s = %s\n", iname, ivalue);
  printf ("> with streams = %d\n", NSTREAM);
  // set up data size of vectors
  int nElem = 1 << 18;
  printf("> vector size = %d\n", nElem);
  size_t nBytes = nElem * sizeof(float);
  // malloc pinned host memory for async memcpy
  float *h_A, *h_B, *hostRef, *gpuRef;
  CHECK(cudaHostAlloc((void**)&h_A, nBytes, cudaHostAllocDefault));
  CHECK(cudaHostAlloc((void**)&h_B, nBytes, cudaHostAllocDefault));
  CHECK(cudaHostAlloc((void**)&gpuRef, nBytes, cudaHostAllocDefault));
  CHECK(cudaHostAlloc((void**)&hostRef, nBytes, cudaHostAllocDefault));
  // initialize data at host side
  initialData(h_A, nElem); initialData(h_B, nElem);
  memset(hostRef, 0, nBytes); memset(gpuRef,  0, nBytes);
```

Page: 6 / 6

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

So, for this example, let us, first see that how things can be I mean, how we can have overlap execution. So, what will first do is we will figure out what is the maximum number of streams that we can have so, we set up this maximum number of streams, because we will be declaring that number of streams and all that so, let us say we first initialize the host side that is on which to work on so, let us say initialize the host side which are getting locked and pinch to the host side memories for that we have support for a synchronous memory copy. And also we perform allocation for this GPU difference and I mean host reference areas here, and then we initialize all these data structures who say that is hA and hB.

(Refer Slide Time: 09:24)



Concurrency Example

```
// malloc device global memory
 float *d_A, *d_B, *d_C;
 CHECK(cudaMalloc((float**)&d_A, nBytes));
 CHECK(cudaMalloc((float**)&d_B, nBytes));
 CHECK(cudaMalloc((float**)&d_C, nBytes));
 cudaEvent_t start, stop;
 CHECK(cudaEventCreate(&start));
 CHECK(cudaEventCreate(&stop));
 // invoke kernel at host side
 dim3 block (BDIM);
 dim3 grid  ((nElem + block.x - 1) / block.x);
 printf("> grid (%d, %d) block (%d, %d)\n", grid.x, grid.y, block.x, block.y);
 // sequential operation
 CHECK(cudaEventRecord(start, 0));
 CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
 CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice));
 CHECK(cudaEventRecord(stop, 0));
 CHECK(cudaEventSynchronize(stop));
 float memcpy_h2d_time;
 CHECK(cudaEventElapsedTime(&memcpy_h2d_time, start, stop));
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

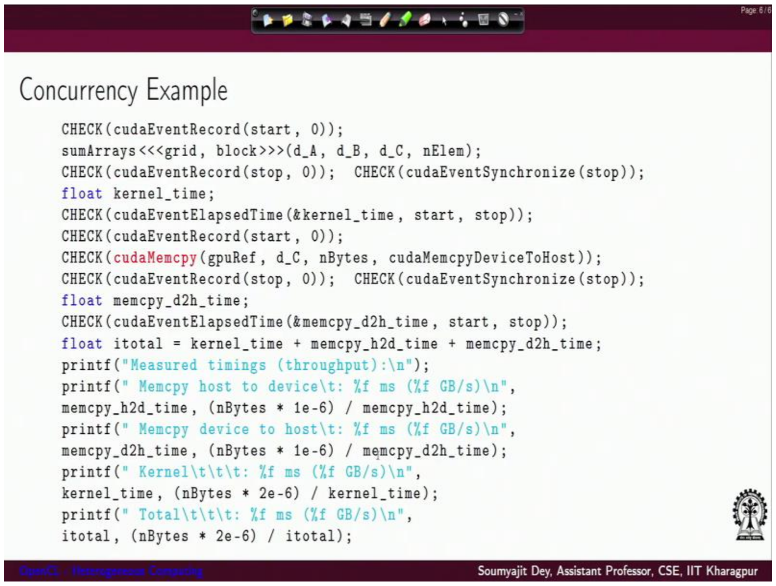
And then on the device side, we actually create the different device that is A, B and C for the individual memory in a device global memory. So, once this is performed, then for bookkeeping purposes we create we define 2 CUDA events start and stop and set them up here for recording. So, we just create the CUDA events and the moment we execute this function CUDA event create So, the timing of whatever is going on next will, they will start recording from this point in the start and stop data structures.

And then sorry so here we just create the events and later on when we will have to CUDA event record that is the point from which the start and stop events will actually start recording the timings. So here we are just creating and declaring the events and creating their corresponding objects. And then let us, we are just performing suitable declarations of the creed and blog dimensions, I mean, using code similar to whatever we have discussed earlier.

And we perform normal memory copy here just to show a normal execution. So you copy the whole side data to the device side data hA to hB. And you perform and before this, I mean copy, you actually start logging the timing using a CUDA event record. In the timer, start in the event start, and here, you record the timing in the event stop. So you actually have the 2 timings before and after the synchronous I mean the normal sequential data transfers.

So, for a normal sequential transport, what is the time elapsed that we will now we can check through a CUDA event elapsed time API by supplying it with this recorded times in this CUDA events at these points in the video will start and stop. So if you supply these 2 time point variables to this API, you get the value reported. And of course, after the CUDA event you will have a CUDA event synchronized call for the stop event.

(Refer Slide Time: 11:42)



Concurrency Example

```
CHECK(cudaEventRecord(start, 0));
sumArrays<<<grid, block>>>(d_A, d_B, d_C, nElem);
CHECK(cudaEventRecord(stop, 0));  CHECK(cudaEventSynchronize(stop));
float kernel_time;
CHECK(cudaEventElapsedTime(&kernel_time, start, stop));
CHECK(cudaEventRecord(start, 0));
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));
CHECK(cudaEventRecord(stop, 0));  CHECK(cudaEventSynchronize(stop));
float memcpy_d2h_time;
CHECK(cudaEventElapsedTime(&memcpy_d2h_time, start, stop));
float itotal = kernel_time + memcpy_h2d_time + memcpy_d2h_time;
printf("Measured timings (throughput):\n");
printf(" Memcpy host to device\t: %f ms (%f GB/s)\n",
memcpy_h2d_time, (nBytes * 1e-6) / memcpy_h2d_time);
printf(" Memcpy device to host\t: %f ms (%f GB/s)\n",
memcpy_d2h_time, (nBytes * 1e-6) / memcpy_d2h_time);
printf(" Kernel\t\t\t: %f ms (%f GB/s)\n",
kernel_time, (nBytes * 2e-6) / kernel_time);
printf(" Total\t\t\t: %f ms (%f GB/s)\n",
itotal, (nBytes * 2e-6) / itotal);
```

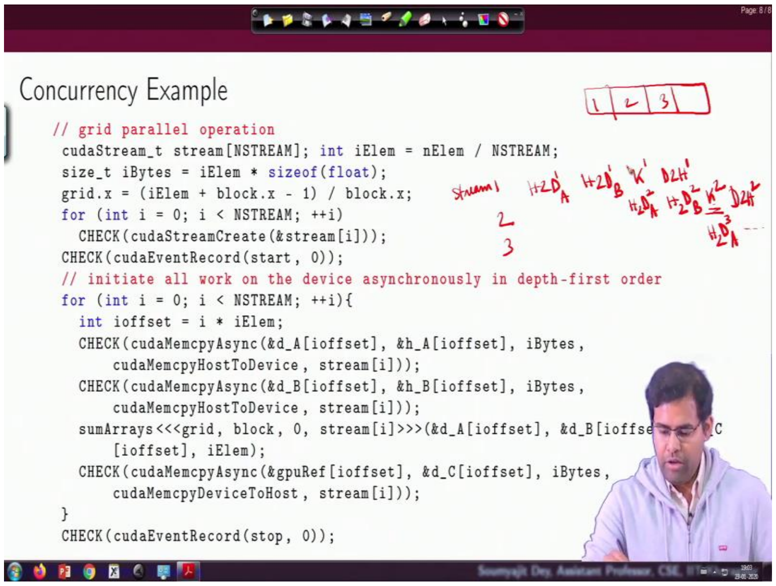
OpenCL - Heterogeneous Computing   Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now in that way, we are just doing a bookkeeping we are just giving an example like how can you perform bookkeeping for all the normal calls? Again, we have a CUDA event record for start and you have a CUDA event record for stop and in between you make a normal kernel call for specific before let us do the sum of edits calculation for that kernel without using the synchronous API.

So, then I mean, we are having everything together in the same program. So, for that we are again performing a CUDA event synchronize on stop here just to ensure that the full stream the default stream here synchronizes on this event and then we can just change the elapsed time and all that now, in case I want to see that what do you I mean, what is the I mean, I would also like to see what is the time required for doing normal sequential copy from the device side to the GPU, CPU side.

We can make again and record a start event and stop event and in between I can copy back the data from the device to host and once that is done and I can compute the elapsed time here and I can print if I mean this measure timings through the print API is here so I can just check what is the normal execution time for this particular GPU device for a normal host to device copy for a normal device to a copy for a normal sequential kernel execution and what is the product time. The reason we are doing this and we are trying to show you usage of this API is soon we will be using this also for the synchronous case and compare the performance statistics.

(Refer Slide Time: 13:26)


1 2 3
Stream 1 $H2D^1_A$ $H2D^1_B$ K^1 $D2H^1$
$H_2D^2_A$ $H_2D^2_B$ K^2 $D2H^2$
2
3
$H_2D^3_A$

Concurrency Example

```
// grid parallel operation
 cudaStream_t stream[NSTREAM]; int iElem = nElem / NSTREAM;
 size_t iBytes = iElem * sizeof(float);
 grid.x = (iElem + block.x - 1) / block.x;
 for (int i = 0; i < NSTREAM; ++i)
   CHECK(cudaStreamCreate(&stream[i]));
 CHECK(cudaEventRecord(start, 0));
 // initiate all work on the device asynchronously in depth-first order
 for (int i = 0; i < NSTREAM; ++i){
   int ioffset = i * iElem;
   CHECK(cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes,
       cudaMemcpyHostToDevice, stream[i]));
   CHECK(cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes,
       cudaMemcpyHostToDevice, stream[i]));
   sumArrays<<<grid, block, 0, stream[i]>>>(&d_A[ioffset], &d_B[ioffse    C
       [ioffset], iElem);
   CHECK(cudaMemcpyAsync(&gpuRef[ioffset], &d_C[ioffset], iBytes,
       cudaMemcpyDeviceToHost, stream[i]));
 }
 CHECK(cudaEventRecord(stop, 0));
```

So now, let us start with a synchronous case. So, we define this number of streams the maximum number of streams which are supported as was gathered from the system earlier. And then for each stream, we start dispatching a synchronous comments. So, we have a loop running from 0 up to the number of streams less than that so that it will run this many number of times. Every time you are creating 1 stream. By invoking the CUDA stream create command for the stream variable that is that has been declared for the stream.

You have a start of you start recording the timing for using the start event variable. And inside, then you mean inside this loop you will define another loop through each now you start dispatching data from the host side to the device side in chunks. So for so in this case, what we are doing is, first you have this loop. I mean, first we have a loop using which you create all the different streams. Then you launch this loop through each for each stream, you start performing a synchronous data transfer.

So for the first stream, you give you provide data from the host to the device with in some chunk. Now the chunk size has been decided by this i offset value, as you can see, so this is actually telling you that well, you want to actually execute. I mean, you want actually that will this stream will copy data of this memory chunk. So just if we want to illustrate to an example, so let us overall memory chunk, and that you want to copy the total size of the buffer.

And let us say that you have you want you want that in each memory copy comment, you copy, 1 chunk of the data, let us say the chunk size is this i element. And in each iteration, you said offset value. I mean by 1 more successive among it just shifted by an amount that is equal to i element. So, you copied it from here to here, then from here, I mean suddenly falling by i element size like this.

And also in each mean in each iteration that you have to specify the number of bytes. So, this is the chunk size in terms of number of elements, of course, you have to specify what is the total number of elements you want to actually what is the total number of bytes these elements occupy. So, that is expected in this I bytes point which we calculated earlier here as you can see, so, number of elements multiplied by the size of the type that is flawed, that is giving you the i bytes.

So, this is a chunk size so, you choose it here and then inside this loop for each of the streams, you are copying 1 chunk of data inside this loop for each of the streams, you are copying 1 chunk of data for input at hA to output a device dA idea input at hB to the device are dB, and so on so forth.. And after these copying from the input data to the device, you also, you know, you make the synchronous call to the launch the kernel on this on the stream.

So, you launch 1 kernel version of some areas, so, to the stream, of course, you want this kernel to work on that specific data chunk that has been copied. So, if we just draw an example, let us say in general is the whole side elements. So in the iteration, we have copied this data and this data to the device side. And so you have 2 of these segments and you want in the istream the kernel launch event will only work on this part in the next stream the kernel launches work on the next part so on so forth like that.

So, essentially you have launched in the full iterations of the loop, you have launched 1 kernel instance in each of the streams and in each launch, the kernel instance is working on that chunk of data that has been copied. So, in that way, you have all the streams engaged with executing some part of the original data or enough amount of data. So, inside this loop, in 1 full iteration of this loop, and that means you iterate through across all the streams.

And each stream is responsible for copying some, I mean some specific chunk of input data. Since there are 2 input areas each stream I mean, in 1 iteration of the loop, you are launching 2 chunks of data from 2 input areas to the device, you are launching 1 kernel instance into the inside that stream to work on these 2 copied instances of chunks of data. And then you are launching another copy comment in the same stream to copy back the data law execute data that is produced by this kernel on this specific stream back to the host side.

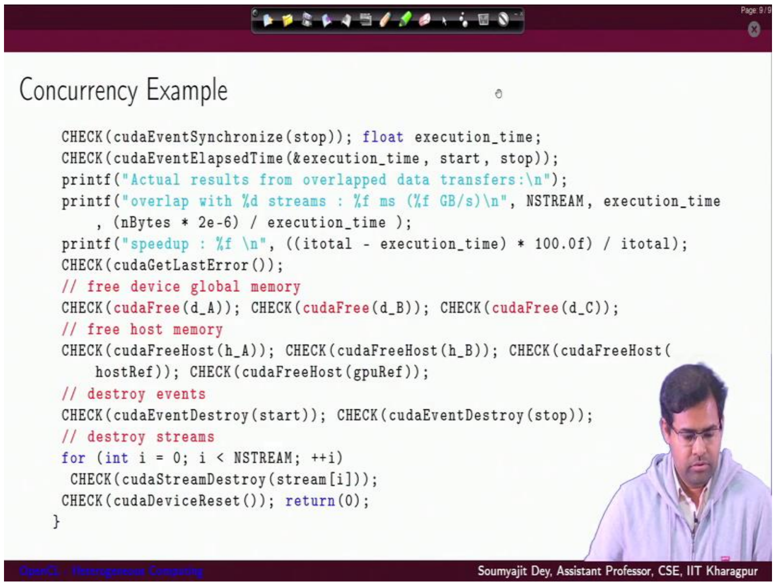
So, all that you are doing is you are breaking the overall copy to the device, overall kernel execution and overall copy to the host operations into small chunks and each chunk is performed the each of the operations of each of our corresponding to each of the chunks are executed in different streams. What will have the advantage will be the pipeline execution that we have discussed earlier that different streams will have some amount of copy some amount of some amount of data to work on, and some amount of copy that.

So they will be able to overlap and they will be able to make good use of the devices. So if I just take an example and draw this figure, so like in stream one, I have an H2D copy of a chunk for array A so let us say this is stream 1, you make a copy of array A some chunk, then an H2D copy of some array B. Then you execute the kernel, so this is the first chunk. And then you have device to his copy of the first chunk. Now, this can of course, overlap with certain operations, we are assuming that the copies cannot overlap.

So here when in stream 1, there is a here we have stream 1, we have the kernel operations, it can overlap with the H2D copy in off stream to have the second chance. So this this index is for the chunk in first stream, you are copying the first chunk. So a chunk would be the first chunk of memory, the second chunk of memory, the third chunk of memory. In each chunk, I have this is a number of elements, the sizes, i bytes and all that so this will overlap.

And I can also have this is should be a second chance. I can have a H2D of array B second chunk, I can launch a second kernel instance here. So this will be the overlapping stream to and then I also have the device to host for the second chunk happening now in the stream 3, I can also have when this kernel execution is going on this is the point where I can start doing the H2D copy of the third chunk in the third stream for the array followed by A, B, and C, so on, so forth. So we have this kind of overlap execution. And that will actually help me to get the pipeline performance benefit like we have been discussing earlier.

(Refer Slide Time: 22:29)



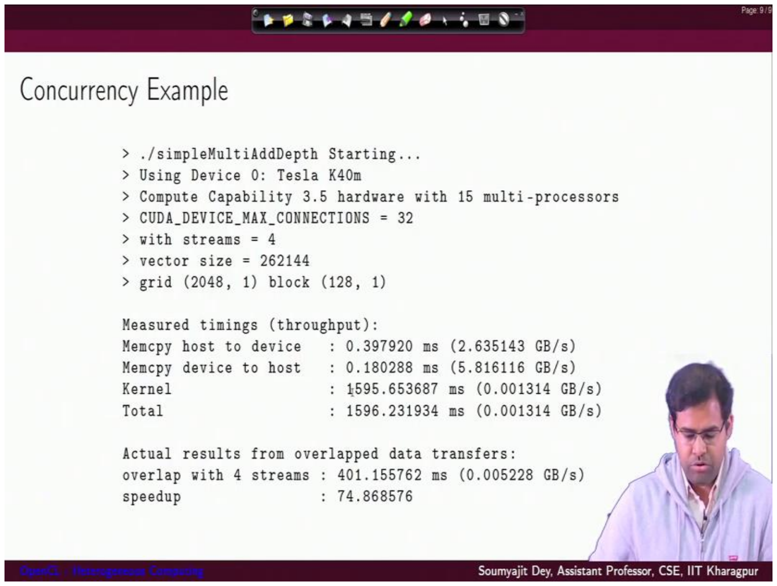
Concurrency Example

```
    CHECK(cudaEventSynchronize(stop)); float execution_time;
    CHECK(cudaEventElapsedTime(&execution_time, start, stop));
    printf("Actual results from overlapped data transfers:\n");
    printf("overlap with %d streams : %f ms (%f GB/s)\n", NSTREAM, execution_time
        , (nBytes * 2e-6) / execution_time );
    printf("speedup : %f \n", ((itotal - execution_time) * 100.0f) / itotal);
    CHECK(cudaGetLastError());
    // free device global memory
    CHECK(cudaFree(d_A)); CHECK(cudaFree(d_B)); CHECK(cudaFree(d_C));
    // free host memory
    CHECK(cudaFreeHost(h_A)); CHECK(cudaFreeHost(h_B)); CHECK(cudaFreeHost(
        hostRef)); CHECK(cudaFreeHost(gpuRef));
    // destroy events
    CHECK(cudaEventDestroy(start)); CHECK(cudaEventDestroy(stop));
    // destroy streams
    for (int i = 0; i < NSTREAM; ++i)
     CHECK(cudaStreamDestroy(stream[i]));
    CHECK(cudaDeviceReset()); return(0);
  }
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, we can, do it like as we have discussed, and then we can once we are out of the loop used to use the stop timer, and then we can actually measure what is execution time. And here we have just the normal freeing events and destroy events like we do for freeing and resetting the device now. So that is one way to perform the concurrent execution.

(Refer Slide Time: 23:03)



Concurrency Example

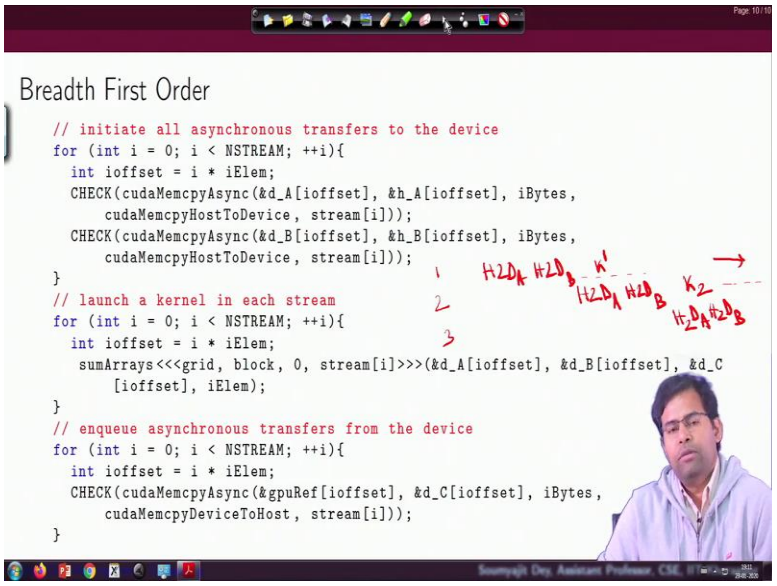
```
> ./simpleMultiAddDepth Starting...
> Using Device 0: Tesla K40m
> Compute Capability 3.5 hardware with 15 multi-processors
> CUDA_DEVICE_MAX_CONNECTIONS = 32
> with streams = 4
> vector size = 262144
> grid (2048, 1) block (128, 1)

Measured timings (throughput):
Memcpy host to device   : 0.397920 ms (2.635143 GB/s)
Memcpy device to host   : 0.180288 ms (5.816116 GB/s)
Kernel                  : 1595.653687 ms (0.001314 GB/s)
Total                   : 1596.231934 ms (0.001314 GB/s)

Actual results from overlapped data transfers:
overlap with 4 streams : 401.155762 ms (0.005228 GB/s)
speedup                : 74.868576
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And then with that we can have certain statistics. So we I mean, there is no point in reading out the statistics is for you to see that what are the execution times for in case you have it I mean, I mean, how much time you have overlaps across streams, what is the effective speed up? How much of overlap as you can see that so, these are the measure timings you have that mem copy time from host to device mem copy time from device to host side for this specific vector size.

So, these are all specific to 40 GPU. So, you can set up maximum 32 streams here, I mean were we are working with 4 streams here. And we are using these vector size we have launched the kernel with this gradient block setting and you have the mem copy times, for the host to device to it, the execution time and all that now, with the optimization of using concurrency by exploiting the streams, you have overlapped the 4 streams to get a reduced execution time of 401 millisecond. So that is the advantage we can see here from overlapped execution.

(Refer Slide Time: 24:17)



Breadth First Order

```
// initiate all asynchronous transfers to the device
for (int i = 0; i < NSTREAM; ++i){
  int ioffset = i * iElem;
  CHECK(cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes,
      cudaMemcpyHostToDevice, stream[i]));
  CHECK(cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes,
      cudaMemcpyHostToDevice, stream[i]));
}
// launch a kernel in each stream
for (int i = 0; i < NSTREAM; ++i){
  int ioffset = i * iElem;
   sumArrays<<<grid, block, 0, stream[i]>>>(&d_A[ioffset], &d_B[ioffset], &d_C
       [ioffset], iElem);
}
// enqueue asynchronous transfers from the device
for (int i = 0; i < NSTREAM; ++i){
  int ioffset = i * iElem;
  CHECK(cudaMemcpyAsync(&gpuRef[ioffset], &d_C[ioffset], iBytes,
      cudaMemcpyDeviceToHost, stream[i]));
}
```

Now, just to highlight that this is not the only possibility, there will also another possibility, what we did where we are copying data in chunks and overlapping the data chunk copies with the kernel execution. But let us say we do it in a order that is for each stream instead of it running across streams and letting each stream copy that data and in chunk let us say, for each stream first, we copy all the data in then for each stream, we execute all the kernels in parallel on the data.

And then for each stream, we copy back all the data. So that can also be an option. So essentially, we are saying that so, here we are, we are also using the streams. But as you can see, we are not overlapping the host to device copies with the kernel execution, but rather we are dividing the copy operation across streams. So, let us say this is my input array and you have stream 1, stream 2, stream 3, you divide the stream into 3 parts.

And you ask stream 1 to do some copy then stream 2 to do some copy and stream 3 to do some copy and in effect you have the entire thing copied. So, in that way with respect to the streams, you do not have much I mean, at this stage, you do not have some advantage because, well, at this stage what is happening is you are asking stream 1 to perform a copy, then stream 2 to perform a copy and stream 3 is performing a copy. But if I look at the scenario that with respect to timing.

And the say now, once these things are done, so, you have the first stream, if you look into the program, so, the first stream has made a copy from the offset up to this size the second stream and this first stream will next perform the copy up to this size let us and then after executing this loop, then it will execute the other loop through which you again have some synchronous dispatch of comments using which you are on each of the streams are launching their respective kernels.

And after that you have another third loop through each you are again executing a synchronous comment for copying bad the data where each of the streams are responsible for copying where certain chunks as per we I mean, as per we are directing here that which stream will copy where with chunk. So, overall, if we look at, well, what will be the overlap executions? So what do we really have? So, you have an H2D copy, let us say we are just trying to write the example.

So, you have an H2D copy of array followed by an H2D of array B and then so this is happening for some stream 1. And then for stream 2 while this is going on, nothing can happen. But now, when this stream 1 has finished the copying for the first and the second arrays, for whatever chunk size it is supposed to do, then the stream to can do its copy. So, it has the H2D copy of the chunk for which it is responsible in array A. And then each 2d copy for the chunk for which it is responsible in array B.

But while this is going on, we can have the kernel for the stream 1 execute somewhere here because there has been by this time this is going on. And this is an asynchronous API. So, you have just launched this comments, then you launch this comments, and then you will launch the copy back comments. So, the host, has just launched all these comments and moved out, this is where the action is happening.

And the CUDA runtime system is trying to manage the executions in the concurrent streams all for the GPU. So here we have sequentialization what so these are launched, but when they are executing, this will be sequentialization. But then when for the next stream, the copying is happening. You can have the kernel execution. And then for the third stream, let us say you have the copies that are starting for then it is chunk H2D for array A, then H2D for array B.

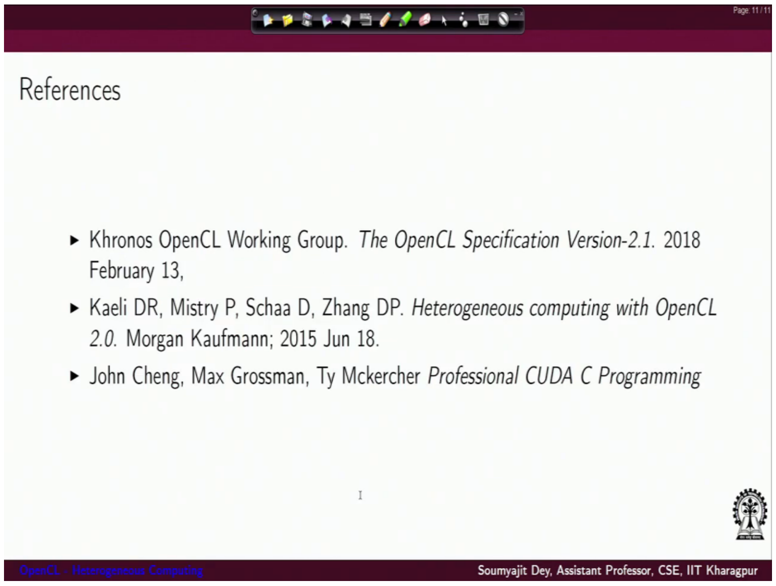
This is the time when the kernel can execute here for the second stream. And at some point of time, I will also have to copy back the data here. So you can go on and construct the rest of the picture here. Like we are just trying to show that earlier, we have been getting overlap execution at our granularity. But here we will be getting overlapped execution at a different regularity. Because you can you do the copies for A and for 1 stream entirely.

And then for the next stream when you do the copy. At that time, you can execute the first kernel. So it is all about how you are actually issuing the comments. And you can either issue them in a depth first order or you can issue them breadth first we are seeing breadth first because as you can see why this is first you break the execution of the original program into streams across the breadth of the program. So, you first kind of you try to submit all the copy comments for the host to device copies, then you try to submit all the kernel launch comments in another loop.

So, you go you go into streaming action and at the second level, once you have covered the breadth of all the streams at the second level, then you go to the third level in the third level, you know covered across all streams the breadth across all streams over a third level. So earlier was the order of streaming concurrency here we have earlier was the depth first order of streaming concurrency whereas here we have a breakfast order of streaming concurrency and we can also have similar execution speed ups here.

So, it really is not the case that we get less speed up in one case or more speed up in the other is just like in what way you want to execute the comments and so that you can have over left execution.

(Refer Slide Time: 31:19)


Page: 11 / 11

References

- Khronos OpenCL Working Group. *The OpenCL Specification Version-2.1.* 2018 February 13,
- Kaeli DR, Mistry P, Schaa D, Zhang DP. *Heterogeneous computing with OpenCL 2.0.* Morgan Kaufmann; 2015 Jun 18.
- John Cheng, Max Grossman, Ty Mckercher *Professional CUDA C Programming*

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

And this will be some of the nice references from which we actually figured out what will be the important text that go into our presentation. Now some we have borrowed pictures as well as a text items from these references. And so these are the open CL and this is CUDA reference is a very nice book on professional CUDA programming. So with this will like to end our lectures for this week. And thank you.