

Parallel Computing
Prof. Subodh Kumar
Department of Computer Science & Engineering
Indian Institute of Technology – Delhi


Module No # 05
Lecture No # 25
CUDA (CONTD...)

(Refer Slide Time: 00:27)

```
__device__ inline void swap(int &a, int &b)
{
    int tmp = a;          // We expect these in register
    a = b;
    b = tmp;
}

__global__ static void ParallelSort(int * values)
{
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;

    // Copy input to shared mem.
    shared[tid] = values[tid];
    __syncthreads();
}
```




Sort Kernel pg 1

Alright so this is where we are we were talking about how the shared memory has been filed and now the sorting has to be happen in this shared memory and this is what is known as bitonic sort which we will talk about. We have not discussed the algorithm yet but very briefly depending on big pattern you swap you compare element who are different from you are in a given big position.

For example in the first iterations 0 and 1 will swap if necessary 3 and 4 will swap 4 and 5 will swap 6 and 7 will swap then you will swap with people to distance away then 4 distance away and so on. And so without getting too much into the details of the algorithm itself you are making all the elements of the thread all the threads of the block going through this loop and each iteration of the loop figuring out which is the bit which at my partner thread is different from me compute your partner threads ID.

(Refer Slide Time: 01:56)

```
// Parallel bitonic sort.
for (unsigned int k = 2; k <= NUM; k >>= 1) {
    for (unsigned int j = k/2; j>0; j <<= 2) {
        unsigned int ixj = tid ^ j;
        if (ixj > tid) {
            if ((tid & k) == 0) {
                if (shared[tid] > shared[ixj])
                    swap(shared[tid], shared[ixj]);
            } else {
                if (shared[tid] < shared[ixj])
                    swap(shared[tid], shared[ixj]);
            }
        }
        __syncthreads();
    }
}
values[tid] = shared[tid]; // Write result.
```

Sort Kernel pg 2

So TID is own ID IXJ is your partners ID which is generated taking XR loop variable J and one of you is going to do the exchange between I and J TID whoever is lower. So if your TID is less than your partner then you do this otherwise the other guy will be figuring out you as a partner and his result will be greater you would be doing this and then you depending on the top level iterate K you either swap if less than your partner or swap if you are greater than your partner okay.

And after everybody has done one swap we have sync because people are moving things around if you want to make sure that all the warp of the block have moved things around so that you can now go to the next level of iteration so you can who is two bits away from me whether I should be swapping with him a lot and was that is done you come out of the loop then you put again your shared all of the threads together you have generated this sorted list and you are going to figure out who puts which element from the shared memory in the local memory and then you have done okay.

After this there is an automatic sync thread because the kernel is done at this point after the kernel is come out all of the block have done. So there is a implicit sync thread you would have to or you figure out some other way to communicate across blocks but there is no sync thread version of across blocks. One is you come out of the kernel then you come out of the kernel of all blocks are done so it is synced.


Other is you can put in some memory location whether this block has modified its data or not we can say every block gets one chunk of memory where it says I have done my part A then I see everybody has done their part A I go to part B. And that is where you would want to do voting within your warp once you have done your block has done your block get you are waiting of other blocks have done their bits.

I will in fact give you another example of something very similar if every block as done their bit then I need to take all the block results and see if everybody has done their bit right. And so all the threads of my block will read the data for every block and you do a voting all is one then all is well alright.

(Refer Slide Time: 05:13)

Memory Fences

- **__threadfence_block**
 - wait until all global/shared memory accesses by the caller are visible to block
- **__threadfence**
 - wait until all memory accesses by the caller are visible to
 - All threads in the thread block for shared memory accesses
 - All threads in the device for global memory accesses
- **__threadfence_system**
 - wait until all memory accesses by the caller are visible to:
 - All threads in the thread block for shared memory accesses,
 - All threads in the device for global memory accesses,
 - Host threads for page-locked host memory accesses (see Section 3.2.5.3).
 - only supported by devices of compute capability 2.x.



Let us continue the discussion that we were beginning to have about synchronization with reference to memory what is the consistency model that the memory is provided now you know what are the hardware does what is that consistency model. So within a block this week because you can say this is a flush and then things before the flush after the flush are ordered across the blocks really no consistency right.

At the end of the kernel but not within so you can think of that has one kind of week where you say that absolutely no guarantee anywhere except at the end of the kernel but it does not count as

week because you do not have a function call that says now we are at the end of the kernel system control and of course you cannot proceed in same context beyond that because the kernel is done at that point.

There is a bit of guarantee in that whenever people write things to memory at the memory level there is consistency in that transactions are not mixed to each other. So memory is seeing a stream of transactions from different multi processors within the multi processors it is going to either discard some of those request or fully perform those request now we have to do part performance of a request okay.

It is not going to mix it is not going to take its address and your data and write it that position either it is taking your address and your data or your address and your data and writing it in the right place. And that is hardware level of consistency without that there is no hope right but you can do something at the software level and for the memories there are these fences that are provided.

There is something called thread fence for a block it is essentially a weaker kind of barrier it is not quite a barrier it is not all threads come and stop here. But when you call this method thread fence block you are guaranteed that anybody who has called this thread fence block in your block is going to see the following behavior everything that they did before the thread fence block is visible to everybody in the block after the thread fence block okay.

So if you wrote something and you have gone past to thread fence block that means I do not know whether the third person come to that block or not but if they do they cannot see what I wrote any I wrote before thread fence block is in the memory guaranteed unless somebody over wrote it that is a different that is a synchronization issue but not all threads in the block in a block who need it would call it this is not a barrier.

It is a blocking flush when you come out of this flush you know that it has reached memory it not just in the process of reaching the memory and there is thread fence which is across blocks right. I when I call thread fence I am ensuring that anybody else running in the same program in the

same kernel is going to see this data after that i have preceded beyond that thread facts. And then there is thread fence system which makes it visible to the host also right. So if I have proceeded beyond the thread fence system host will see this data now there is any other questions and thread is only on CUDA threads.

There are other two are available in the earlier version also with in a warp is essentially like thread fence rather sync threads I think stronger it is definitely processor consistent but I am thinking that even stronger than that because you know precisely the instant where all this calls were made right so you order them you can make them sequentially consistent yeah actually you are write what happens is if you are doing it depends on the axis it if your warp is writing in this when it is sequentially consistent.

Because in fact even for warp half warp definitely but even for warp it is sequentially consistent because a half warp entire read is going to happen in one shot or right whatever it is going to go to the memory in single transaction and two half was because you know exact what happens after which also get fully ordered. So each half warp entire memory transactions goes in one order if it is something bigger than in it you are reading 8 bytes per element then even that can be done in one transaction.

But something much bigger that cannot be handled in one transaction then it is going to get partition into multiple memory transactions. But even that actually is deterministic that algorithm is consistent hardware every time makes the same decision and it is public. So as a programmer you also know what is among the different read and write calls to the memory and because they are going to go through the same cache also there is going to be absolutely reordering anywhere.

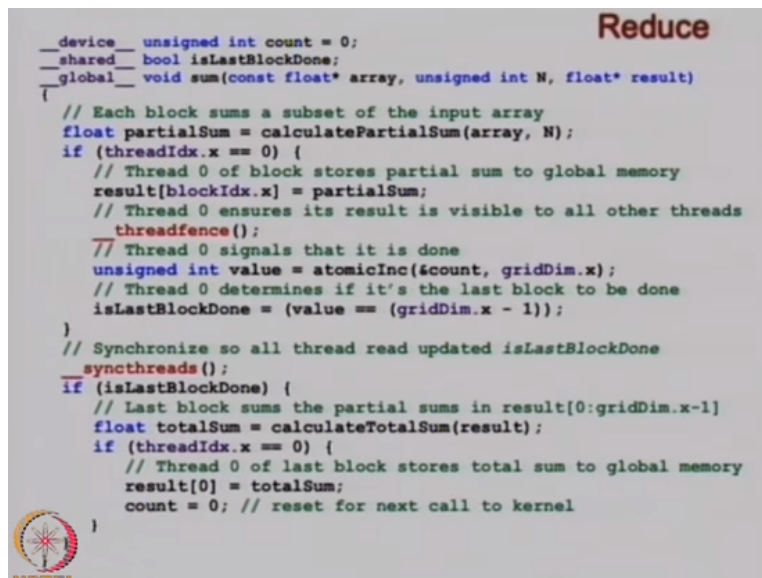
So that is not true so it is definitely sequentially consistent now that I think about it on one dot X 2 dot X made it in order to make it more general has caused some consistency up do not need sync threads all the time even in 1 dot X. It depends on whether you are syncing across warps or not I do not know what is exactly what you are referring to but sync is essentially ensuring that all the blocks all the warps of the block have come to this position and you for me nothing this about it.

All the blocks are executing in their own order and it is possible that these block this warp of the block has reached instruction number 10 where that warp of the same block is at instruction number. So unless you sync are going to get to the same spot it there are multiple warps of the blocks reading data just like in our example we saw we needed to sync thread that is not going to remove for 2 dot X.

Because every warp in the block has to have finished its read before you can proceed further so if you as a warp has to depend on some other warp who have finished something before you can proceed further then you have to wait for him you have to sync with him and 1 dot X or 2 dot X both of them will have to do the same thing okay so that two we were discussing and 1 dot X because of no reordering in the cache there was sequential consistency in 2 dot X because they are letting cache lines evicted in their own order it is V consistency.

You can still make sure that you get flushed but you just by writing you are not going to get there. You can use volatile if you only use volatile then you get back to having a sequential consistency okay.

(Refer Slide Time: 17:11)



```
device unsigned int count = 0;
shared bool isLastBlockDone;
global void sum(const float* array, unsigned int N, float* result)
{
    // Each block sums a subset of the input array
    float partialSum = calculatePartialSum(array, N);
    if (threadIdx.x == 0) {
        // Thread 0 of block stores partial sum to global memory
        result[blockIdx.x] = partialSum;
        // Thread 0 ensures its result is visible to all other threads
        threadfence();
        // Thread 0 signals that it is done
        unsigned int value = atomicInc(&count, gridDim.x);
        // Thread 0 determines if it's the last block to be done
        isLastBlockDone = (value == (gridDim.x - 1));
    }
    // Synchronize so all thread read updated isLastBlockDone
    syncthreads();
    if (isLastBlockDone) {
        // Last block sums the partial sums in result[0:gridDim.x-1]
        float totalSum = calculateTotalSum(result);
        if (threadIdx.x == 0) {
            // Thread 0 of last block stores total sum to global memory
            result[0] = totalSum;
            count = 0; // reset for next call to kernel
        }
    }
}
```

So let us look at that piece of code which uses very small font size and also third fence. So let us begin with where we are talking about memory fences right so there was synchronization where

you said explicit synchronization between different threads of the block and that is really all the synchronization that is provided among CUDA threads between kernels there is some synchronization right.

So one kernel clearly comes before another kernel and unless you want them to run concurrently they will run in sequence and so everything that has happened in the previous kernel will finish before the next kernel begins most of the time which means that unless you want otherwise.

There are then so that is the synchronization side of the story the other side is indirect synchronization which is making sure that is written in memory are actually visible fences for that and this is something that you would see even in multi core CPU thread fences are there you may not directly use them or you may not typically have functions that in turn call the memories thread fence function of the primitives that the hardware provides.

But the function cause like lock malloc etc will typically use those things under the hood all right so the thread friends we talked about is on the block which says that I have written something and everybody in the block now knows about it I have written something and everybody on the thread on the device knows about it and I have written something and even the system knows about system knows about it was long as what I am writing to is also visible using mapped memory on those sides all right.

So this is where we are talking about that code which I have now split into two slides but it seems a little small but I think it is readable now right. So it is reduce you have a block which takes the element that are within the purview of the block the responsibility of the block acts it together and then multiple blocks are adding the elements within their and then somehow they need one of them or multiple of them need to them take these partial sums that each block is producing and find the grand sum find the total sum.

And so sum of the details are abstracted over here you begin by the block so this is the kernel right so for a block there are sharing this is the data somehow where computing the partial sum using whatever technique that they may have used which takes the input which is the list of

elements and generate the sum for this block partial sum is for the given block would you declare it has partial sum for a given block is that declaration.

I meant is it shared device any qualifier necessary not the type right now who owns it each thread as its own partial sum okay. Now if I am thread index 0 hopefully everybody is reaching the same partial sum in the shared memory everybody is doing the log reduction right. So everybody has their own partial sum and one data needs to be produced for this entire block. So I will say one of the thread 0 takes the responsibility and it writes its partial sum which we are going to assume is the same on every thread which is the sum of the block okay.

So if thread index is 0 then you take your partial put it in your block ID there is a result which stores partial results for each block result array and there is one slot result for every block you figure out your block ID you could that is your slot you put your sum there and nothing fence happening so far. Then why do you need a thread fence? So this is you are writing once for a block one member of the block is representing for the block writing for the block right.

And ah that item may be cache they may be sharing which means also cache over there for some other block which may or may not be running on the same multi-processor. So they may be sharing but that fall sharing more to do with efficiency So let us proceed what is going to happen let us in fact for talk about what should happen everybody figure out their partial sum and somebody is going to add the partial sum.

You need to make sure you when that somebody is going add the partial sum or a group of threads is going add the partial sum is that everybody has figured out partial sum and written it right. The blocks are independent of each other they are not synchronized so your block is done and you have written your partial sum it does not say anything about where the other blocks are they may not even begun yet.

You ensure here that whatever you have written is going to go into the memory beyond this point so anybody else who reads it who is going to it who is going to combine the elements within the result array. And whoever reads it may not be you if you read it you are going to get it without

even thread sense if somebody else reads it you need have thread fence here. So that they can get updated or the correct value that you are actually wrote here and it is not setting in cache something okay.

And then there is how do you that everybody is written so you can go to the next phase and add the partial sums. In this case there is a counter and everybody has written atomically incrementing there is a single count that says how many blocks have done you know how many blocks are working on this grant problem right. And you say I am also done somebody else say I am also done so you are implementing by one a global counter okay.

And those of who did something of this sort should know what this atomic is doing raise their grid dim dot X there it is incrementing by one this variable concurrent which is in the global memory none of you used atomic serializes at a very short interval but sometime you have no choice here you have a choice if you were not doing atomic what would you do? What is one of the thing or a few of the thing you might be.

So return the kernel is one but that had it is overhead right now another kernel will have to start and do something but what was the other suggestion you enable your bit my bit is done. But how do you know that all bits are done when and who will do that some one of the blocks job is to figure out whether all the grids are one that can be done but it is little expensive so here there are all this computing options and depending on how slow the atomic is it may or may not be faster than say keeping a block doing nothing but checking it all bits are one right.

So the second parameter here says that the maximum that you can implemented to okay it will map to a single TDX instructions which is supported in the hardware it effectively mean you read something you add something to it and you put the value back this instruction always implements one and there is max it says do not go about it in fact it resets it to 0 when you reach there.

So you can have a recycle instruction okay so you have implemented your value and somebody needs to check come value has become the full size of the grid how many blocks you have

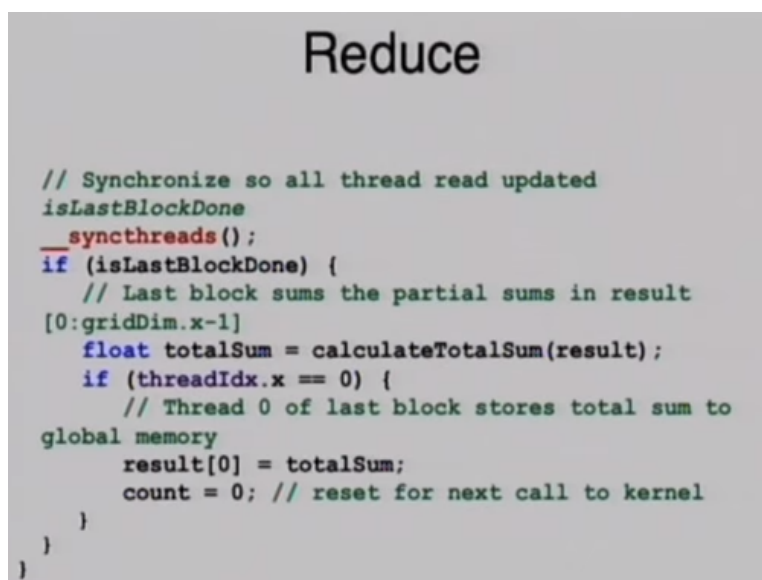
started with so everybody does that so once you have done with your partial some you can go check if that things is the accounted has reached max. So everybody is parallel is figuring out is everybody else done once they have done okay.

Because it will never reach that value it has to go to 0 right you start with 0 and keep incrementing and you will grid dim dot X is number blocks the value returned is the old value when you do atomic you get the old value back right not the value you do are incrementing 2 so everybody is figuring out again it is a local this variable is local to this block it is in shared and so you have figured out this variable has reached and all this blocks are figuring out whether this is done right.

Once you have checked that you come out so blocks who reached their early are not going to go back and check in this case they are not busy waiting there is not loop here the way thread index is 0 which means I am representing my block to write my result I am going to check I am going to implement this count when check if the account was already at $N - 1$ other people have written it before me and going to set my variable to is the last block done.

So how many people will see this variable to be true on thread of one block the block that happens to get the last access to the atomic operation.

(Refer Slide Time: 31:53)



```
Reduce

// Synchronize so all thread read updated
isLastBlockDone
__syncthreads();
if (isLastBlockDone) {
    // Last block sums the partial sums in result
    [0:gridDim.x-1]
    float totalSum = calculateTotalSum(result);
    if (threadIdx.x == 0) {
        // Thread 0 of last block stores total sum to
        global memory
        result[0] = totalSum;
        count = 0; // reset for next call to kernel
    }
}
}
```

Then we do a sync threads why is that necessary they are but what does it mean all threads of the block has found that it was the last block are going to in parallel some partial sums so their they are going to become active now and start participating and summing it right but the threads if you look back here did not go through if condition right. So they immediately went through the end should they wait for this guy to check and update whether this is the block to do the partial sum or that the final sum of course they should right.

They are waiting their until this thread come along and says yes we are the last our variable which is shared across all of us is set to true meaning that this is the last block to come and let us now find the final sum okay. So now everybody checks is last block done and if this is the block that is the last block and then all threads of this block will set will see this to be true and now they are going to go figure out the sum in this partial result array and then if I am thread 0 of this block.

I am going to write the resulting value just like the previous when I was doing within a block I am going to write the result in value total sum into the result okay. So this demonstrates that where you I think at least hope they are some threads of was obvious you had to wait there to figure out whether or not you were responsible for doing the remaining work and the thread fence.

Whenever you want other blocks to read what you have written in this case other blocks are somewhere on the device but when you have written is in the global memory and accessible to other blocks then you make sure the thread fence is done. So that it does not sit stale in the cache somewhere okay questions on memory synchronization alright.

(Refer Slide Time: 34:41)

Streams

- **Functions in a stream execute in order**
 - A stream is akin to a higher-level “thread”
 - Different streams may interleave (subject to mem ops)
- **To use streams:**
 - create a stream object
 - specify it as a parameter to
 - kernel launches
 - host ↔ device memory copies
- **No stream => 0-stream**
 - 0-stream is a “serial” stream
 - Begins only after all preceding operations are done
 - No subsequent operation may begin until it is done.

So finally let us also talk about how to increase the concurrency by using stream when we want to have multiple kernels running in parallel or having memory transfers is going on for one kernel when some other kernel is actually executing and so on and so forth. So the idea stream is straight forward you break down your computation into different sequences each still is sequential from the host point of view.

But these different sequences of stream so to speak are concurrent in not independent of each other because they are some restrictions on when a given stream operations can run versus other stream in fact rules will go through an example to see those kinds of restrictions and I will also show you the examples of how a stream is run by code. So you do not worry about a exact sequence that is being demonstrated in the slide one thing to note is that there always streams.

When you are not explicitly using a stream you are saying everything is going to stream 0 right form host point of view there is sequential stream called stream 0 and when you do not specify a stream or your operations are not explicitly target towards the stream then if they are targeted towards streams and everything on stream 0 is sequential when stream 0 is running no there stream is running other stream must wait for stream 0 to finish and so if you actually want concurrency in places where you want concurrency you will not be sending things to stream 0.

You can also query of few things about the stream you also have this notion of events so you can say whether this stream has reached a certain point and so on and hence so forth.

(Refer Slide Time: 36:53)

```
Example Using Streams

cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2*size); // pagelocked memory
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i*size, hostPtr + i*size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    DoKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i*size, inputDevPtr + i*size, size);
    cudaMemcpyAsync(hostPtr + i*size, outputDevPtr + i*size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
    Later cudaStreamDestroy
```

But let us look at the example you would declare some variable of type CUDA stream and get allocate a few streams so it could a stream create as many stream as you like again there is some maximum limit. In this case a page lock memory host stage is allocated using CUDA malloc host of certain size then you do inner loop what you would normally do in a stream in stream what we do you do you say memory copy some memory from host to device run this kernel get the result back.

That thing that ripple has been quoting a loop so now you are saying for hint equal to 0 to 2 which is 0 and 1 you do this thing but at the so in the CUDA memory copy sync the last parameter says which is the stream for which this CUDA operation holds from the target and so the first memcpy kernel memcpy back is going to stream 0 not the sequential stream not the stream ID is 0 whatever is in your stream variable here stream array 0 which will move certainly not be 0.

And then do the same sequence for stream 1 and later if you wish you can destroy the streams can something go wrong. In this case probably nothing goes wrong because if you look at what the memory areas they are copying it is totally based joint first guy says my left half do the

kernel on that part then my get the left half back second one in the mean time doing the same on the right half and hopefully there is no dependency on left and right now cannot know that depends on the kernel.

So what is going to happen is that two mem copy has long as the older devices everything is synchronized so memcopy is and kernel launches are all going to happen independently which is 1.0 I think 1.1 onwards memcopy and kernels can overlap but under more gradually less restrictive set of condition. So for mostly any CUDA device find today the first two copies can happen in parallel.

So your left half going into the CUDA and right half going into CUDA device can happen in parallel will it happen in parallel here. Can it happen in parallel here whatever you structured it as you said copy for copy the left half then launch the kernel for the left half then bring the results back now copy the right half okay it may be the sync half right which means that you have returned fine.

But can you as a system implemented allow the second to host copy to device to proceed that is the thing that is why I said they are not independent you cannot assume they are independent you could right then entire responsibility goes to the program to make sure that whatever the interaction are happening within stream are fully protected by mutual exclusion or some synchronization method.

It is their concurrent which means it is more of a feature that use a program can use to speed your computation you can have multiple kernels running so the enough thread available for every SM to have some warp to run at every block that is the ultimately what would you like there is schedules it can be that one kernel is running on first three SM's. And the second kernel at the same time running on the forth SM it does not help you to tell right where if it is possible to run it.

The system will run it possible meaning there are this dependencies there are few dependencies that need to be respected if you respect those and there is device available on which new blocks

can be scheduled will be scheduled global memory is the same. So if your kernels the two kernels you have launched interfere with each other then you can get in trouble yes precisely it can happen when that means you are sharing across two different concurrent kernel cost to different think of it on the CPU side two different thread sharing a shared memory.

GPU is yes always be occupied more kernels right you want to do something independent at the same time or even if independent as long as you make sure that the sharing is properly controlled you can run it right so what CUDA will not allow you to do is run a new kernel where the old kernel results are not back from the memory okay. And so in this old kernels result which has been ordered earlier by a program order.

There is a host program order the host program order will be respected by certain aspect of the stream okay. One of them is that if the data has not come back then ant kernel launch or in fact any CUDA copy that is host ordered later cannot appear first okay. So that defeats this entire concurrency in different threads you can do it in different host X then this dependence is lost but if you wanted this dependency you keep it there if you do not want this dependency you can put it two different hosts.

Which by the way did not work very well implementation issues for older versions of CUDA I believe until do you have any. I though in CUDA until 2.0 or even something you did not try before that there was some issues with multiple host threads running in parallel which is not there anymore so that is one way the other would be you put the three statement in their own loops right.

What will that do? That will say that two main copies are after each other the two kernel launches are after each other and then the two results are after each other so now the two main copies can happen in parallel kernel launch is can be launched in parallel and the results can be copied back in parallel. There is a bit of a restriction still that the copying cannot being until the last block of the kernels have been lost are actually being sent okay.


So that slightly reduces the amount of concurrency that you would otherwise get earlier blocks launching can begin launching is not an atomic operation right you can here is some more work and GPU will at some point say I can only take on that much work right now and so if you have big kernel launch which has lots of blocks then until the last block is launched you cannot start reading the results coming out okay.

So that is the restriction that is the reason for a different stream you cannot start copying back stream once result back until 0 is whatever last block has been launched okay.

(Refer Slide Time: 48:59)

Stream Synchronization

- **cudaThreadSynchronize()**
 - waits until all preceding commands in all streams have completed
- **cudaStreamSynchronize()**
 - waits until all preceding commands in the given stream have completed
- **cudaStreamWaitEvent()**
 - makes all the commands added to the given stream after this call to wait until the given event
- **cudaStreamQuery()**
 - check if all preceding commands in stream have completed



Of course within your own stream you have to do that all right there also as we just discussed synchronization allowed or procedure for synchronization there is CUDA thread synchronize which is it is more like fence it has the notion the semantics very similar to fence where you say I say CUDA thread synchronize then all the preceding streams or commands that are issued to preceding stream this is again from the point of view of a single post thread okay.

When you multiple thread then they are concurrent independent to each other within a thread all the stream cause you have made has to be finished. Stream synchronize is a version of thread synchronized except it takes a parameter which is all the calls of that particular stream have to be finished okay. And then there is a CUDA stream wait event which takes a stream and a given

event and it says all the calls for that stream up to that event have to be finished and then there is a query about the problem of stream right you have fold of many stream from your thread.

And we want to know how far they will be reached so then you can query for the streams assuming the stream are marking either counter which exist or events which tell you how to monitor the stream or how to read the progress that you are making okay so that is as much as we are going to talk about the features of CUDA I still want to spend a little bit of time talking about the performance issues like how many threads and how to read memory in all that any questions and I am going to stop here.