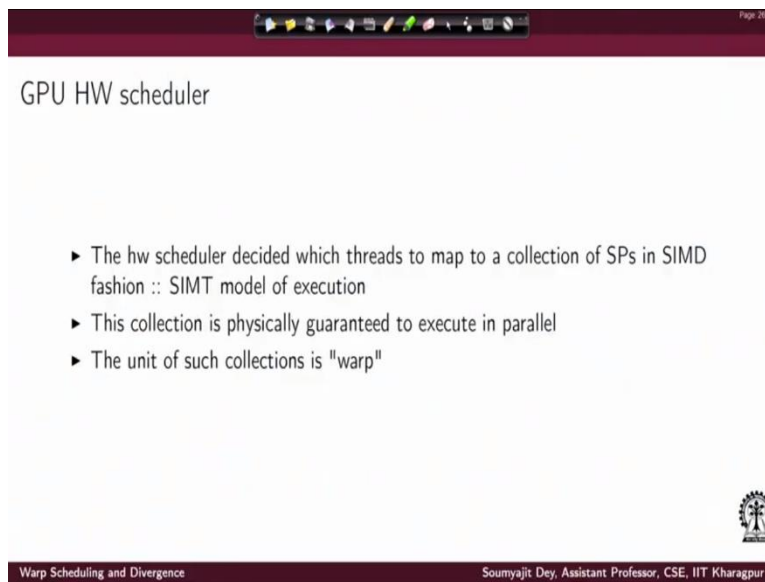


**GPU Architectures and Programming**  
**Prof. Soumyajit Dey**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology – Kharagpur**

**Module No # 04**  
**Lecture No # 17**  
**Warp Scheduling and Divergence**


Hi, So, welcome to our lecture series on GPU architectures and programming. So in the last assignment we were discussing about GPU hardware schedulers and definition of a warp. So (00:37) we will start from that point.

**(Refer Slide time 00:40)**



GPU HW scheduler

- ▶ The hw scheduler decided which threads to map to a collection of SPs in SIMD fashion :: SIMT model of execution
- ▶ This collection is physically guaranteed to execute in parallel
- ▶ The unit of such collections is "warp"

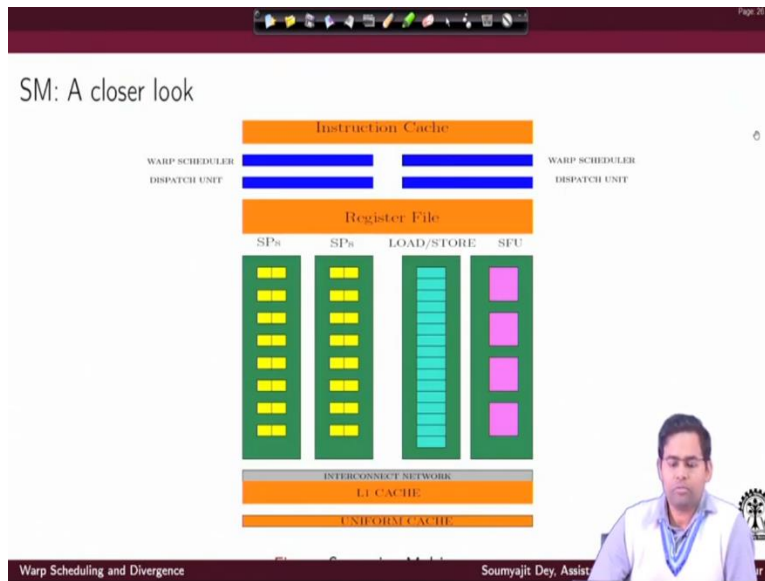


Warp Scheduling and Divergence      Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So essentially when we map kernels launched set of threads or a GPU hardware. So we have a hierarchy of hardware schedulers which will now come into play. So at the high level there will be 1 component of a hardware scheduler which will decide which thread block gets mapped to which of the SM's. And at the lower level inside each of the SM's there would be another component scheduler which will decide which of the threads of the map thread blocks will constitute what we call as a warp.

So warp will essentially be a collection of such threads which execute in parallel as discussed earlier. So this packing of threads from thread blocks to warps is something that will be decided by the lower level scheduler

**(Refer Slide Time 01:30)**



And if we just re loop into SM so we have this kind of warp scheduler sitting inside the SM and there is a high level scheduler which is distributing blocks into SM's for execution.

**(Refer Slide Time 01:44)**

**Warps**

- ▶ Warp is a unit of thread Scheduling in SMs.
- ▶ Warp size is implementation specific (typically 32 threads)
- ▶ Warps are executed in an SIMD fashion i.e. the warp scheduler launches warps of threads and each warp typically executes one instruction across parallel threads.

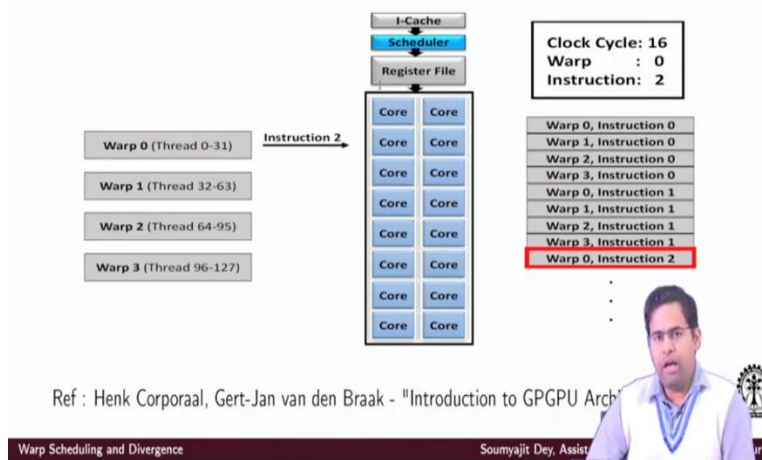
Ex : If a SM has 128 SPs, it can execute 4 Warps at a given time (one Warp has 32 Threads )

A small video inset of the presenter is visible in the bottom right corner of the slide.

And so just to summarize again what a warp is it is a collection of 32 threads which will execute for using the SIMD instruction sequences and the warps get mapped into the collection of SPs for execution. So if I SM has an example 128 SP's then it can execute 4 warps at any given time. Since 1 warp has 32 threads so at any time in parallel the SM we will have 4 warps proceeding through its SP's in parallel.

**(Refer Slide Time 02:15)**

## Warp Scheduling in SM



So here we have an example scenario where we are trying to provide situation that ok we have an SM where as you can see that we have a collection of SP cores. So here you have in total 16 SP cores right. And consider the situation that you have some warps progressing to this SM. So for example let us say warp 0 its constituting now this packing of threads into the warps is what is going to be done by the scheduler which is sitting inside each of the SM as we discussed that this is the second level scheduler in the hierarchy.

The high level scheduler is dispatching thread blocks to each SM and inside this is SM we have this kind of scheduler who is actually forming this warps. So this low level SM scheduler is actually forming this warps like the form of warp 0 warp 1 so this kind of hiding each warp and it is deciding in each warp which are the threading indexes that should be constituting each of the warps.

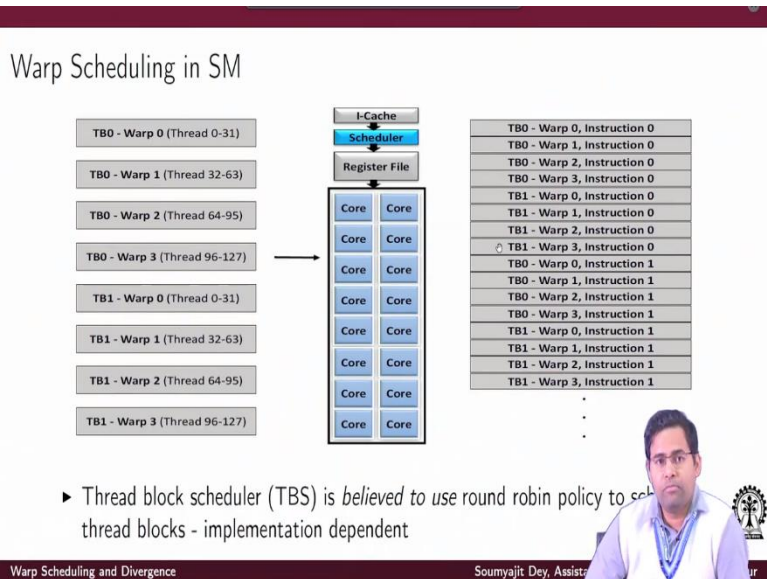
So I have warp 0, warp 1, warp 2, warp 3 and they are containing threads with ID 0 to 31, 32 to 63, 64 to 95 and so on so forth and they are executing the instructions that are part of that is that and that CUDA kernel under question that instruction sequence. So just to look into this scenario on the right-hand side we are trying to say that what is the exact sequence of this warps that are executing. Suppose I have an instruction sequence that is instruction 0 followed by instruction 1, followed by instruction 2.

So if in this hypothetical system hypothetical lesson I am executing this warp as you can see that I have a collection of a this 16 codes and each warp is containing 32 threads right. So assuming everybody take a single cycle of execution for each of the instruction on the core there is no floating pointer delay due to other kinds of complex operation. Each warp would then complete with 2 clock cycles right because there are 32 operations to execute in parallel for the warp while I have only 16 cores that are available.

So the warp would technically take 2 clock cycles. So in that way for instruction 0 to be executed by all the 4 warps as you can see here I would be I would be consuming. So each warp is consuming 2 clock cycles. So for this 4 warps all of them completing instruction 0 I would require 8 clock cycles. Similarly I would again require 8 clock cycles for instruction 1 and this was the execution precedes here.

So again for warp 0 executing instruction 2 that would start in the 16 clock cycle. This is just an example scenario we are trying to give. So we are trying to say that this is how the warps may get scheduled here on the core and the number of cycles that the warp would take depends on the number of SP's that are available. And also the kind of instruction that the warp is executing.

**(Refer Slide Time 05:33)**



Now if I have a relook into the picture when I am considering warps belonging to different thread blocks. So I am now considering warp 0 from thread block 0 and then warp 1, warp 2, warp 3 all from thread block 0 and then again warp 0, warp 1, warp 2, warp 3 all from thread

block 1 right. So now I have 4 warps. Their 4 of them belong to thread block 0, 4 of them belong to thread block 1.

And if we map their execution here on an SM with 16 SP cores that is like earlier I would have each of the warps executing in two clock cycles. So I would have for example here instruction 0 getting executed by all the warps. Warp 0, warp 1, warp 2, warp 3 belonging to thread block 0. Then instruction 0 executed by warp 0, warp 1, warp 2, warp 3 belonging to thread block 1. And similarly, with instruction 0 completed by all the thread block warps I have instruction 1 again getting executed for warps belonging to thread block 0.

Then again I have instruction 1 getting executed for all the warps that are belong to thread block 1. So in this case of course we are having this assumptions that there is a thread blocks scheduler which is using a round robin policy to schedule the thread blocks. Now this is what we have as an open domain answer in the academic research papers that people are assuming that the NVIDIA thread block scheduler is following this kind of round robin policy.

But of course, it depends on your implementation these are the exact hardware scheduling strategy for the thread block schedulers as well as the warp schedulers is not, they are in the open domain. And of course, it can also be implementation dependent depending on different scenario somebody can implement the GPU hardware in a different way to have a different possible scheduling of thread blocks.

From a programmer's point of view, we will always assume that the warps can proceed at their own speeds. And of course, they have to satisfy the instruction sequence requirement that is enforced by the thread.

**(Refer Slide Time 07:57)**

## Warp Scheduling in SM

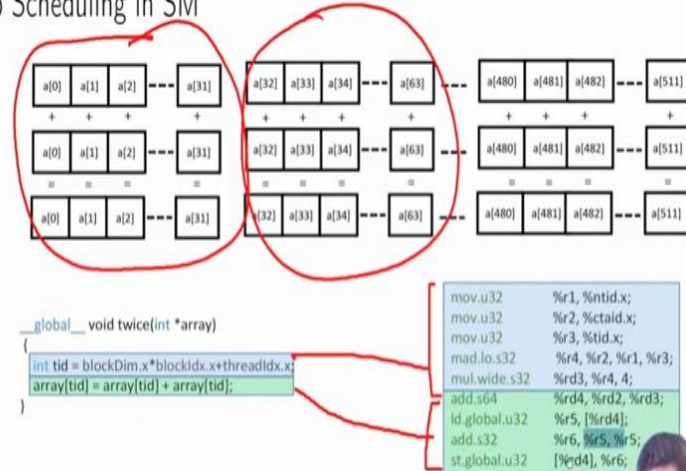


Figure: Simple CUDA Kernel



So just looking into some more examples for here we have a very simple kernel where we can see that the kernel is 2 just look at the 2 line code segment on the left hand side. So we are just computing the thread ID and we are then using the thread ID to access some specific location of an array and simply adding the value in the location with itself and storing right there back. And on the upper part of the figure we show that how the warp will pat the operations.

So in warp 0 I would have the locations a 0 to a 31 getting updated sequence of operations. So this is what would get done by warp 0 and then next we would have warp 1, warp 2 and so on so forth. And so this is how I am just trying to show how the warps would be packing this addition operation spread over the width of the data space. Now if we take a closer look in to the corresponding PTX instruction for the actual CUDA code we have on the left-hand side.

So on the right hand side do we have the PTX instruction. So the initial part of the instruction they correspond to the thread ID computation part and the second part of the instruction they actually corresponds to the editor operation that has been done right. So just take a second and have a closer look in to the instruction sequence here. So essentially as you can see that the first 5 instructions which are highlighted in blue they are responsible for doing the thread id calculation here.

And finally in the second highlighted part we are using this value of the thread ID to bring the content of the array at that location corresponding to the thread ID into this register here. So

essentially this is your final thread ID here and then you use the thread ID to load the corresponding locational value using this load instruction here in r5. And then you are just adding up the content of r5 and storing into r6. And then you do a global store back in to the location for the same thread ID right.

**(Refer slide Time 10:45)**

### Warp Scheduling in SM

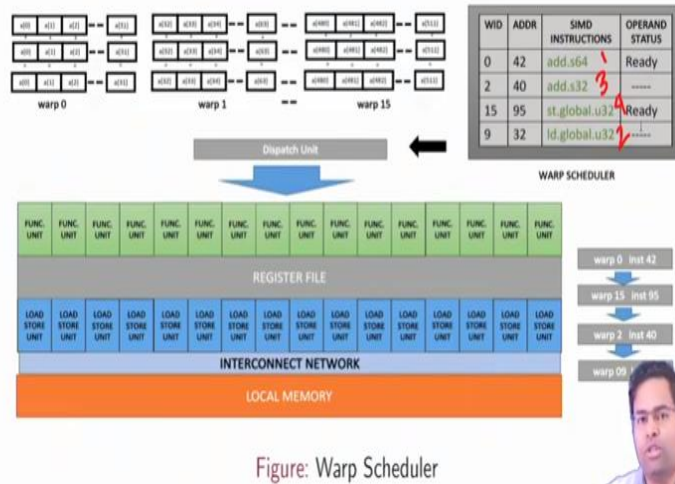


Figure: Warp Scheduler

So if we now have a look into that sequence of instructions that are executing there. So as you can see you have an address followed by a load operation followed by another add operation and then a store operation. So for this different packing of warps like warp 0, warp 1, warp 2 this instruction the corresponding instruction would be dispatched to the SM the different functional units of the SM and I mean how really would their execution be ordered.

So of course for this the warp scheduler need the operands for each of the instructions to be ready right. So there would be a mapping table from which the warp scheduler gets to know that what is the warp ID which is ready to execute so for example these are samples space situation we are showing here that warp 0 is ready to execute from this address 42. And the corresponding SM instruction that needs to be executed is this add is 64 which is basically the first instruction.

So if we just number the instruction for that part this is the first instruction if you again have a loop. The next add is the third instruction and between you have the load and then the store. So the original sequence of the instruction is as I have marked in here. And as you can see what we are trying to communicate here is a different warps may have be they are ready state for different

possible instructions depending on what are the operands available for that instruction whether they are ready.

And also for that warp whether the preceding instructions have already executed. So here we have one possible valid execution sequence for this instruction. So the warp 0 would execute so that is your instruction with address 42. So basically this is the warp 0 is executing the first instruction corresponding to this sum right. So this is the first instruction corresponding to sum operation warp 0 is executing that.

Now this is followed by warp 15 going to execute the fourth instruction corresponding to the sum operation. So oh how can this be possible well of course that would mean that warp 15 must have already executed the previous instructions corresponding to the sum operation for each part of the data space. Now in this example we have warp 15 followed by warp 2. So essentially we are saying that somehow warp 15 made progress at a higher speed and then warp 2 executes its instruction number 3.

That means by this time warp 2 as executed its first and second instruction and then warp 9 would be executing its corresponding instruction which is the second instruction in the sequence. So the idea is very simple for every warp need to follow the instruction sequence 1, 2, 3, 4 as has been marked for this is the first instruction, second instruction, third instruction, fourth instruction. This is a situation we are trying to show here.

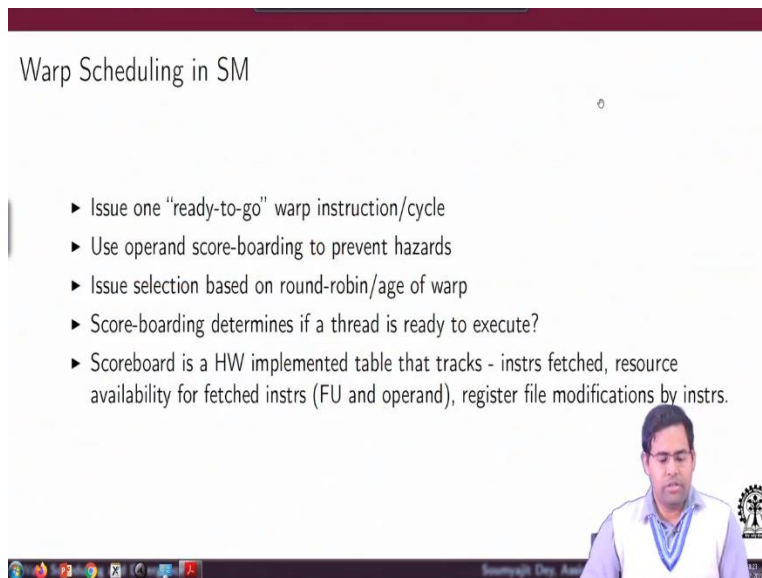
So warp 0 is going to execute first instruction followed by warp 2 executing the third instruction is just an example we are trying to say that ok. So when warp 0 is executed the first instruction warp 15 is executing the fourth instruction that would mean that the warp 15 has already executed instruction number 1, 2 and 3. Now the requirement is every warp needs to execute the instruction first, second, third, and fourth in the sequence.

But it is not necessary that warp 0 would execute instruction 1, 2, 3, 4 followed by warp 1 executing instruction want to support like that. Now there it is necessary that warp 0 execute instruction 1 and then warp 2 execute instruction 1 (()) (15:04) like that. They are free to proceed depending on as and when the operands are ready. So just like an example we are trying to say that it is not only the case that the warps would be schedule following the specific policy it is



also the situation that the warps required the corresponding operands values to be readily available for making some progress right.

**(Refer Slide time 15:32)**



Warp Scheduling in SM

- ▶ Issue one "ready-to-go" warp instruction/cycle
- ▶ Use operand score-boarding to prevent hazards
- ▶ Issue selection based on round-robin/age of warp
- ▶ Score-boarding determines if a thread is ready to execute?
- ▶ Scoreboard is a HW implemented table that tracks - instrs fetched, resource availability for fetched instrs (FU and operand), register file modifications by instrs.

So just to summarize that how does really warp scheduling take place in the SM. So the warp scheduler will issue there ready to go warps one warps instruction per cycle. And it will use an operand score boarding to prevent hazards. So this is kind of that table we are trying to show here just as an example that what are the warps which are ready to go. Now I mean which ones are ready to go they actually are being computed by using a score boarding technique we are not getting into too much of detail of that.

But of course you need to understand that just like the pipe line here also I need to prevent hazard that means I should not be executing a warp before the operands become readily available or I should not issue a warp before the functional units become free right. So these are the potential hazard scenario's which needs to be prevented. For that we will be using the score boarding approach.

Now the issue selection is based on round robin I mean there can be specific examples there can be a round robin scheduling of warps as an example we discussed earlier. However that need not be the only system it can also be based on how old is the warp active for how much time. So it depending the age of the warp and of course the warp has to be ready. That means the corresponding operands need to be available and functional units need to be available.

So the score boarding determines whether in that way the thread is ready to execute and essentially it is a hardware implemented table which tracks different scenario like whether the instruction has really been fetched whether the resources required by the instruction are really available. That mean whether the functional units are available whether the operands are ready and whatever is the register file that is going to be modified by this instruction whether that can also be done.

So the score board actually takes care of providing a status that whether executing this instruction would be hazard free. If so then it would say that well this instruction is ready to go with corresponding to operands and functional units. And then the warp scheduler will use its implementation algorithm whether it is round robin or some other technique to decide whether to issue the warp warps instruction or not.

So this is how possibly the warp scheduling we executed. Of course some part of it is implementation dependent but these are the current techniques that are in actual implementation in modern GPUs.

**(Refer Slide Time 18:17)**

### Latency Tolerance

- ▶ When threads in one warp execute a long-latency operation (read from global memory), the warp scheduler will dispatch and execute other warps until that operation is finished.
- ▶ Other long latency operations : FP units, Branch instructions
- ▶ After all, all threads in the same control-flow execute same instruction sequence on different data points !
- ▶ A common practice is to launch thread blocks of a size that is a multiple of the warp size to maximally utilize threads.
- ▶ Slow global memory accesses by threads in a warp may be optimized using coalescing (more on this later)



Now the other important thing is that why really do we have this kind of warp-based executions and why do we really have this kind of a multiple SP so many in the presence of SP's in very high number inside the SM's. The reason is that you want the GPU's to tolerate long latency

operations. So when threads in one warp executes such long latency operation for example a read operation from the global memory as we all know that read from the share memory is much faster with respective read from the global memory which is very slow.

So whenever a warp would require any read operation from the global memory it has to wait for a long amount of time. Now since I have this warp scheduler internally built into the SM. So it can actually when this warp is waiting for the operands to available the warp scheduler can dispatch and execute many other exit warps until the operands are available for the warp that was waiting right.

Also a warp has to wait may have to wait for many other long latency operations like floating point operations branches and that. So overall the issue is that if a long latency operation is there then the GPU should be able to hide the corresponding penalty then the way to hide the penalty is you always keep your functional units engage by using some other warp. That is why the warp scheduler has a huge role to play in terms of feeding all the SP's with as many operations possible by dispatching warps.

Whenever some executing warps stalls due to a long latency operation. So after one has to remember that all threads that you have which are following the same control flow that means following the same sequence of its and same sequence of instructions they are essentially executing the same instruction sequence. So it does not really matter which threads execute first or which threads inside in another warp executes slow.

As long as they do not have any dependency among each other because they are all doing the same instruction sequence computation on different on possibly different data points on the memory. So also a common practice in this regard is to launch thread blocks of a size that is multiple of the warp size. Now this is something we have also discussed earlier that if we launch a thread block with size which is multiple of the warp size then I have all warps completely filled up with 32 threads.

Otherwise if I have a thread block size which is not a multiple of warp size then I will have some warps which are not really full with 32 threads and that would be that would actually transfer to some hardware not getting utilized while execution of that warp. And also there is important

question like whenever the thread in a warp are executing a long latency operation like a global memory read.

The warps can be optimized in a such a way that the threads can be written in such an optimized way. That the warp will always access consecutive global memory locations. So that the fetch can be done in parallel this is known as global memory coalescing optimization which is something we will take up later on.

**(Refer Slide Time 21:43)**

### Efficient use of thread blocks

#### Target System Constraints

- ▶ A maximum of 8 blocks and 1024 threads per SM
- ▶ A maximum of 512 threads per block

Table: Solutions for various block scenarios

Input Block Size	Blocks per SM	Threads per Block	Remarks
8 * 8	12	64	SM execution resources will be underutilized
16 * 16	4	256	Achieves full thread capacity in SMs
32 * 32	1	1024	Exceeds the limit of 512 threads per block

$$\begin{array}{r}
 64 \\
 12 \\
 \hline
 128 \\
 64 \times 2 \\
 \hline
 768 \\
 256 \\
 4 \\
 \hline
 1024
 \end{array}$$



Now the next thing is how to make a good efficient use of thread blocks. So we take a simple situation here considered some target system constants. So you have a maximum of 8 blocks and 1024 threads per SM. And you have a the maximum number of blocks threads allowed inside block or the thread block size let it be 512. So you are going to have 1024 threads per SM you can have maximum 8 blocks.

What ever you choose as a thread block size and whatever you choose as a number of threads that you launch in total we are restricting that 1 SM can handle 1024 threads and 1 SM can handle 8 blocks it is just as synthetic scenario we are assuming these just an example here. And let us also consider that a thread block size is limited to 512. Now in this situation consider different possible input block sizes for some kernel launch parameters.

Consider the input block size as 8 cross 8 and so you have a block per SM as 12. And so and then in this case so if your input block size is a 68 cross 8 and you have blocks per SM as 12. And then you have threads per block since you are having a this blocks per SM as 12 and input block size is 68 cross 8. So you naturally have a straight for block as 64. So in this case our observation would be that the execution resources in the SM would be underutilized.

Now the question is why that so? So in this case you are actually using the number of blocks per SM is 12 right. While you have threads per block as 64 now the problem would be so since you have the 64 blocks 64 threads per block. So when the SM would be executing you have a in total a thread block size of 64 and in total you have 12 blocks. So overall what would be your total number of threads that you are essentially launching.

So overall you have launched essentially 768 threads right. So ideally I would say that this is I mean the SM could have handled more number of threads. So execution resources are practically underutilized. Now consider another situation where you have got this a input block size 16 cross 16 2D definitions. Now you have you are going to launch 4 blocks in the SM and the number of threads that you are allowing per block in naturally 16 cross 16 which is 256.

So your total number of threads you are launching is. So that essentially is a number which that SM can maximally handle. So in this case you are using the full capacity of the SM right. Now consider a third scenario suppose where use you are defining a kernel with a block per block parameters 32 cross 32 an you are just launching 1 block. So then as you can see 32 cross 32 is also 1024 so essentially you are defining a thread block of size 1024.

Now this example we have taken it violates the definition that I am not going to allow here for this specific example more than 512 threads per block. So this exceeds the limit so this is not allowed. So these are the typical problems that you can have while mapping a kernel to a GPU you have to know it is hardware's limitations. So as we have discussed earlier that there are such specific limitations and you need to know them to define the corresponding different parameters spaces and corresponding launch parameters for that count.

**(Refer Slide time 26:35)**

## Querying Device Properties

CUDA API provides constructs for obtaining properties of the target GPU.

- ▶ `cudaGetDeviceCount()`: Obtains the number of devices in the system.
- ▶ `cudaGetDeviceProperties()`: Returns the property values of a particular device



Now something about different ways in which a GPU can be queried. So for I can the CUDA library provides you several specific constant this kind of constructs using which from your program you can actually figure out what is the hardware configuration of the target GPU. For example in your program you can use this function called CUDA Get Device Count to give you the number of devices in the system.

You can use this function called CUDA Get Device Properties which we again would return different possible property values for some spark particular device in the system. Now why is this important because as we discussed earlier your programs kernel launch parameters may be decided based on this different properties that you get. And also I can have so that would mean when I write my CUDA kernel I would have the parameters of the kernel may be certain access expressions in the kernel in term of variable which would get initialize based on the different CUDA device properties that I can (( )) (27:46) through this kind of library function process.

So with this we will conclude this lecture and from the next lecture maybe we will take a deeper look into the different querying different ways in which device properties can be required. Thank you.