GPU Architectures and Programming

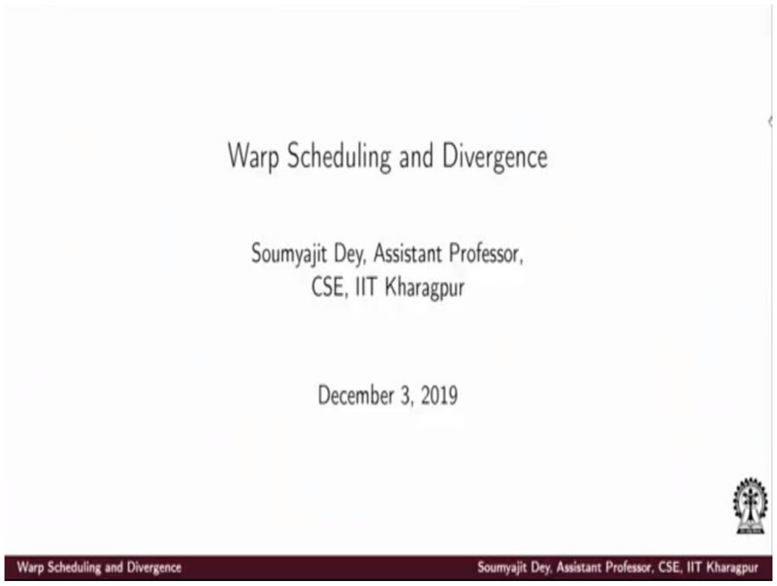
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 04
Lecture No # 16
Warp Scheduling and Divergence

Hi everybody so in the last lecture we have given a summary of how multi-dimensional data can be processed by a CUDA programs and how threads inside a thread block can synchronize among each other. While discussing that we also talked about I mean of course we have discussed this earlier also that in GPU program execution there is specifically for CUDA program execution we have this concept of warps.

So basically that would mean which are the threads that are going to execute together in lockstep at least from the programmers point of view even not even from even if not from the hardware's point of view. So we will from this lecture we will discuss in more detail with respect to that how the GPU essentially is going to schedule the threads in packets of size warps and what are the performance issues that can actually occurred if the programmer is not aware of the way the program is going to be scheduled.

(Refer Slide Time 01:26)

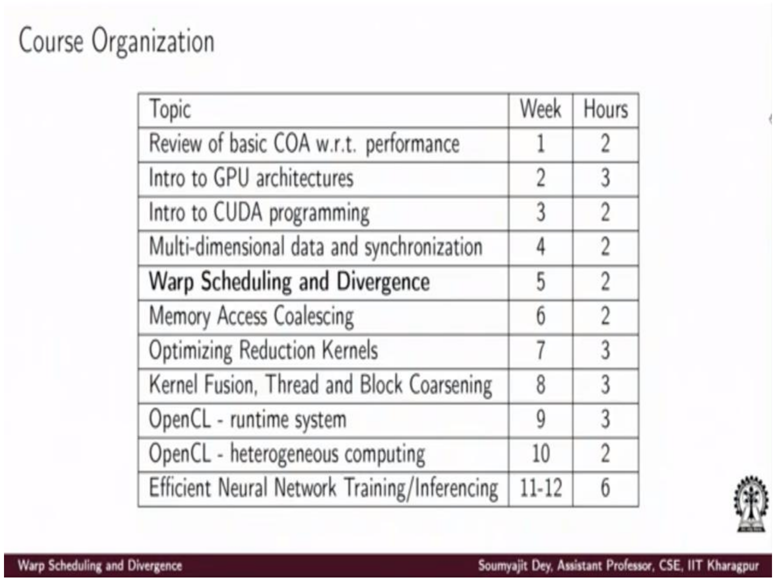

Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

December 3, 2019

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

(Refer Slide Time 01:29)



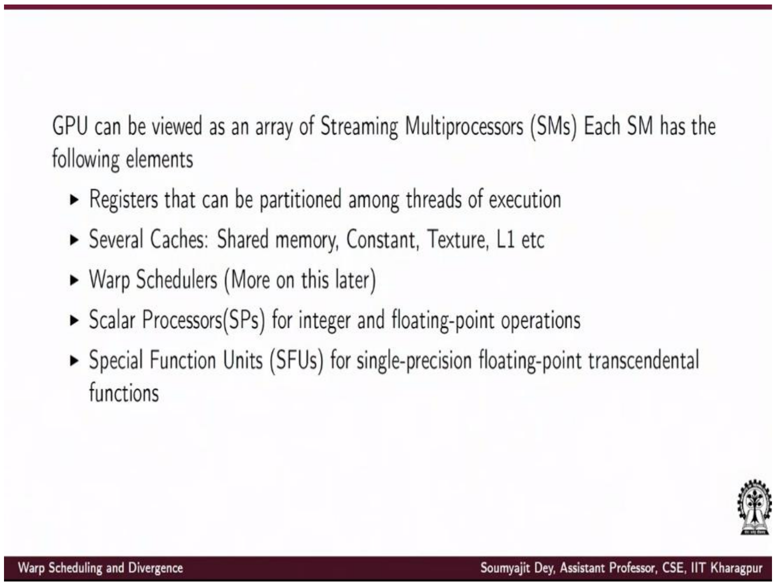
Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

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

So just to do a summary here so that the topic is warp scheduling and divergence and this is our week 5 topic here.

(Refer Slide Time 01:33)



GPU can be viewed as an array of Streaming Multiprocessors (SMs) Each SM has the following elements

- Registers that can be partitioned among threads of execution
- Several Caches: Shared memory, Constant, Texture, L1 etc
- Warp Schedulers (More on this later)
- Scalar Processors(SPs) for integer and floating-point operations
- Special Function Units (SFUs) for single-precision floating-point transcendental functions

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

And so doing a summary of overall GPU architecture if you remember that the GPU can be viewed as an array of a streaming multiprocessors where each SM the streaming multiprocessors has a following elements. It has got a large register file that can be partitioned among the threads for execution. So it is basically portioned across the SMs and I mean the SPs the scalar processors inside the SMs.

The inside the SM apart from having a large register file you also have this several types of cache. For example you have a piece of memory segment which can be partition to behave as a shared memory and also a part of it to be as an L1 cache referred to our earlier lectures on GPU architecture for this. And also inside the SM you have got separate cache memory area for storing constant value to be used inside the program. The constant cache and also the texture cache.

Also each SM inside it has got the specific piece of hardware which is called the warp scheduler which is going to decide which threads are going to execute when inside the SM. The actual execution units comprised the scaler processors which contain integer ALU and the floating point unit along with I mean as a separate unit not as a part of the scalar processor. You also have special function units present inside the SM which are responsible for computing the transcendental functions for example trigonometric function.

(Refer Slide Time 03:07)

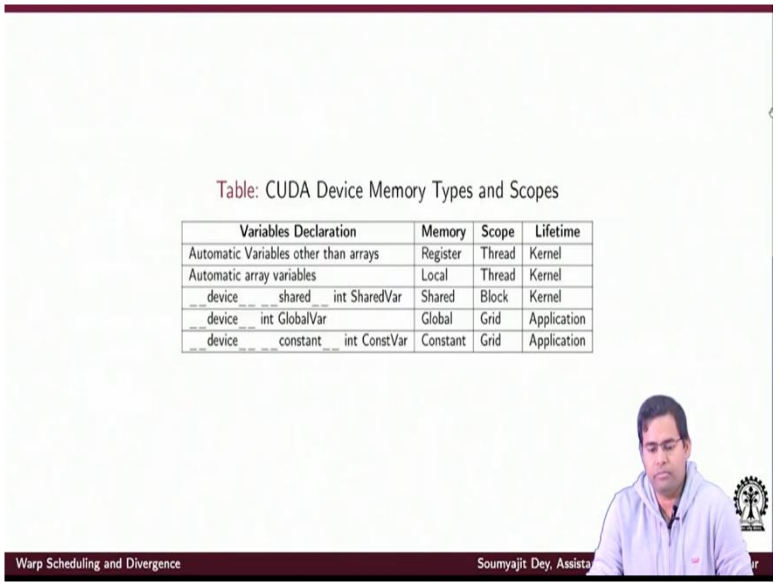


Table: CUDA Device Memory Types and Scopes

Variables Declaration	Memory	Scope	Lifetime
Automatic Variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
__device__ __shared__ int SharedVar	Shared	Block	Kernel
__device__ int GlobalVar	Global	Grid	Application
__device__ __constant__ int ConstVar	Constant	Grid	Application

Warp Scheduling and Divergence

Soumyajit Dey, Assista

Now just to review the different data types and you use inside your CUDA programs and their corresponding scopes that get decided based on the way you in which you declare the variable. So as along as your variable is not an array variable is a it is an automatic variable it is the scope is a thread. So it is specifically it is going to be stored in the register. It is not an array variable it is going to be stored in a register as long as the register is available its scope is part thread and the life time of the variable is the execution of the kernel.

So if you have a local variable V which is going to be used by every thread. So I every thread will see own copy of this variable and with the corresponding mapping in one of the registers right. If you have an array variable for that also the scope is thread every variable will see its own version of the array variable if it is if it is declared inside the kernel as an as local array variable.

The memory type is local that means it will be stored in the segment of the DRAM which is result for the execution of this thread of course since it is part of the DRAM it is a local memory in terms of in the parlance of the programmer but physically it is located far away so the access is slow again the life time is that of a kernel. If is declared as shared then the place where this variable will be mapped physically is the shared memory segments of the SMs.

The visibility will be the block that means all the threads inside a same executing thread block will have consistent view of this variable which is defined as a shared variable the life time is again the kernel that means the moments the kernel finishes its execution the this variable scope is lost right. And then if we declare a variable as a global variable if we declare the variable as a global variable then I mean the way you declare is simply have it as a variable type as a device type variable without the annotation of shared or something.

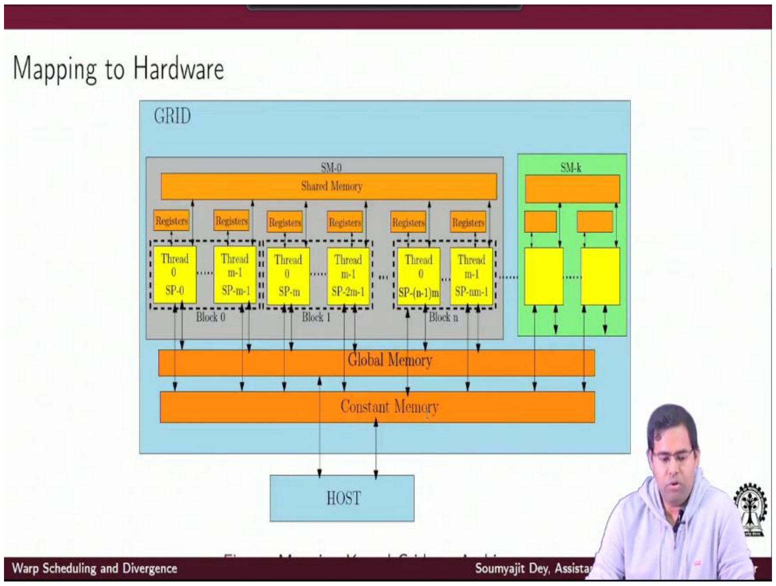
So that would actually infer it as a global variable. It will be it will be resident in the global memory and then the scope is actually grid. That means the variable is alive across the different I mean across the entire execution of the application that is it is alive across multiple kernel instances as operated from the host programs site. If the variable type is defined as constant, then its location is in the specific constant location constant memory location of the memory hierarchy of the GPU.

Again the scope is grid and its lifetime is the application that means it is consistently accessible by multiple instances or multiple different kernels. All of them are assumed to be control by the same host program. So one application is one host program so one host program can comprise multiple kernel launches. So for such scope that is grid the variable has to be a type global or constant. Its definition should for constant it should have a constant quiver.

If nothing is there it is infer type is device. Then infer that is a global variable type is device and you have the shared connotation then you have this defined as a shared variable. But with shared variables as we know that that the scope is the block all the threads inside the thread block will have a consistent view. Lifetime would be kernel and also for normal array and non-array type variables depending on whether it is array then memory is locals, if it is non array a single instance then the memory is register for both of them the scope is thread.

They are every thread has a different copy of them located in the local memory or the local memory essential with the private memory corresponding to the thread in the DRAM or the corresponding register that the thread has been allocated. And again the lifetime is kernel. So this are the different memory types and scopes that we can define for the CUDA device memory site. So as you can see that these are all I mean I mean as long as it is a part of the shared or global or the constant type memory the scope is specified explicitly with this device connation here fine.

(Refer Slide Time 07:59)

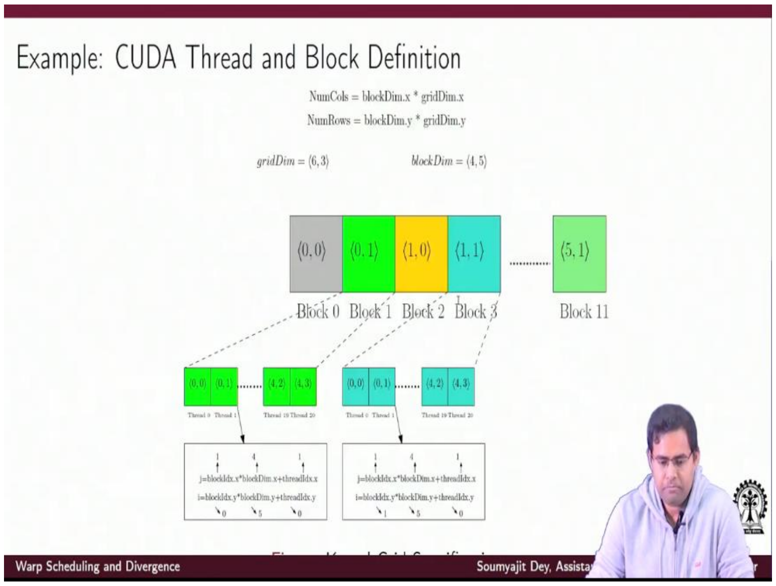


So with this progress now the question that comes is how really do thread get mapped to the hardware. So this is an example picture that we are trying to give here. So we are trying to say that we suppose I have defined threads in a way that I have blocks and each thread block contains m number of threads. So we are trying to show a sample mapping here were we are trying to show multiple SMs.

So this is SMs 0, this is SM k in between we have with the dots we are trying to represent there are lot of other SMs in between. Inside each SM we are trying to give a figure that how the register file is logically partitioned for the SP’s that means for the thread to do the computation privately some significant some part of the register is block for each thread to do their computation store their automatic variables and all that.

And each thread is mapped to each of the SP’s and so if I have a thread block then the threads in the block are getting mapped to one specific SP like this. We are just trying to show a possible mapping. Now of course the question may come suppose I have more than this m number of blocks or let us say I have more blocks then they are SP’s available inside an SM. So then what will really happened.

(Refer Slide Time 09:27)



Example: CUDA Thread and Block Definition

NumCols = blockDim.x * gridDim.x

NumRows = blockDim.y * gridDim.y

$gridDim = \langle 6,3 \rangle$ $blockDim = \langle 4,5 \rangle$

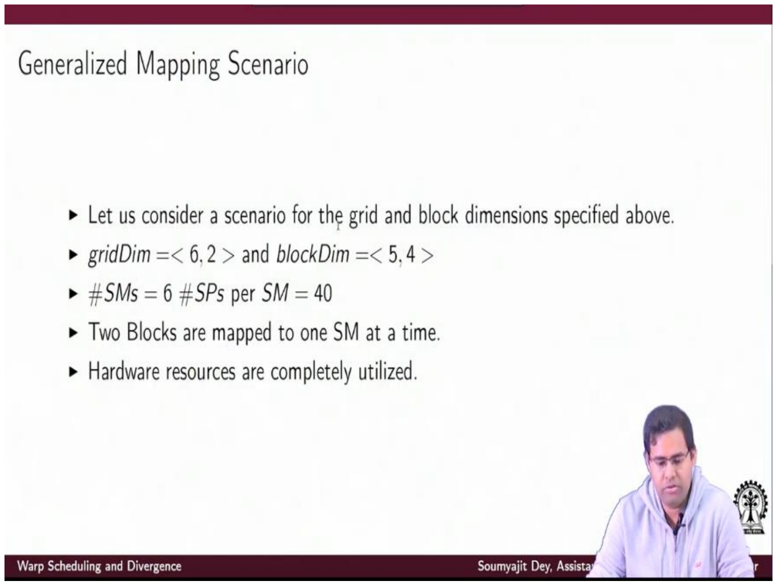
j=blockIdx.x*blockDim.x+threadIdx.x

i=blockIdx.y*blockDim.y+threadIdx.y

Warp Scheduling and Divergence Soumyajit Dey, Assista

So before going to those question just we are trying to keep an example we are trying to recall one of the example mappings as we discussed. So this is one of the example of 2D mapping here. So we are trying to show that we have this many blocks in the 2 dimensional space for each block we have again threads which have been which have been mapped into the blocks like this.

(Refer Slide Time 09:54)



Generalized Mapping Scenario

- Let us consider a scenario for the grid and block dimensions specified above.
- *gridDim* $=< 6, 2 >$ and *blockDim* $=< 5, 4 >$
- $\#SMs = 6$ $\#SPs$ per $SM = 40$
- Two Blocks are mapped to one SM at a time.
- Hardware resources are completely utilized.

Warp Scheduling and Divergence

Soumyajit Dey, Assista

So this was one of the earlier example we are just again here for example purpose and consider a generalize a mapping scenario here. So you have got this 20 threads inside this block and you have got total how many blocks you have got in total 18 blocks right. So you have 18 blocks and 20 threads here. So or maybe we may consider the mapping scenario we have set of grid set of blocks and the block dimension let say 5, 4 so that is you have 20 20 threads per block.

Consider that you have a 6 SP’s SM’s in to together and the total number of scalar processors per SM is 40. So we are considering a scenario where you have an SM considering containing 40 SP’s and you have threads thread blocks of size 20. So what does that mean that would mean that 2 blocks are mapped to 1 SM at a time. Why is that a good thing because your hardware resources are completely utilized.

Why is that so? The reason is you have to you have to remember this as a rule of thumb that a block cannot be partition while mapping across SM’s. A block cannot map half of the block to SM 1 and half of the block to SM 2 that is not allowed. So as long as you have blocks the thread blocks size which is like I mean you have so as long as the scenario is like this that you have the block dimension and the number of SP’s in the SM is the multiple of the block dimension.

You can block and you can have an integer number of blocks getting mapped inside the SM’s without any SP line are utilized so that is the good mapping. So with that example mapping if you just instance share the earlier picture this here everything was symbol so we are considering

M number of threads and this M threads where getting mapped to this I mean N number of threads per block and we are considering N blocks inside an SM.

So in this example what is happening is we are having 2 blocks right. So we are having 2 thread blocks which are getting mapped to 1 SM here each thread block is containing 20 threads right just to recall here we have discussed here there would be 40 SP's per SM and in total for each block there are 20 thread. So that would mean for the 40 SP's we can logically partitioned them across to in sets of 20 and we can map each block to one of the collections of 20 SP's right.

So I have block 0, 0 mapped to this nice collection of 20 SPs. I have block 0, 1 mapped to this nice collection of 20 SP's like this for SM 0. Overall if I am considering 6 SM's and then I can go on like this and map the other blocks and since I have 6 SM's and I have got this many blocks this many thread blocks to map. So I should be able to map I should be able to keep on mapping like this 2 thread blocks per SM.

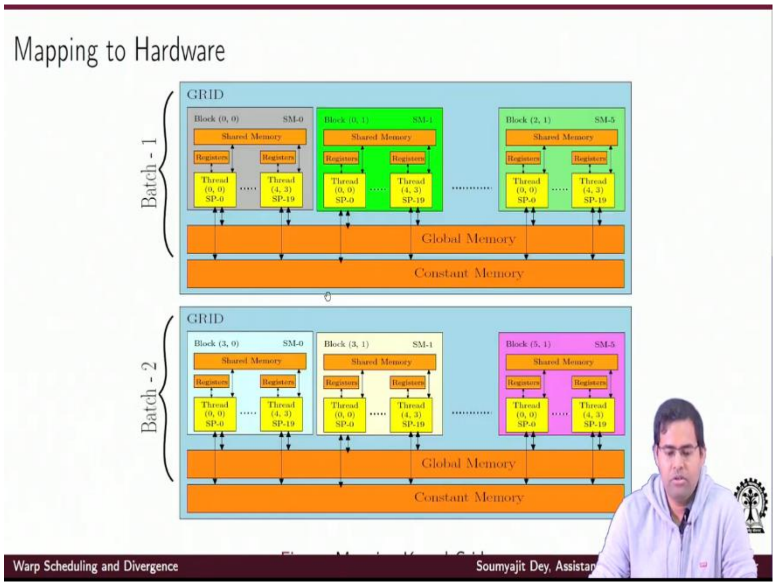
And in the way I would be having an execution of how many blocks as you can see 2 blocks are getting executed per SM. I have got 6 SM's so overall, I have 12 blocks executing in parallel right inside this entire GPU system. Well what about the other blocks? I had 20 of them the scheduler is holding them back because they are (()) (13:47) enough CUDA codes available execute all of the thread blocks in parallel now which is quite general scenario.

Your kernel dimensions launched parameter dimensions allow you to launch a large number of threads. Of course, physically you may not have that many code available across these hierarchy of SM's but that does not matter. The hardware scheduler takes care of mapping this thread blocks inside this SM's in kind of this is an example packing I am showing in in with this kind of mappings.

And ones some thread blocks will finish execution some other thread block which actually get map into one of the SM's and this were the execution will continue. So consider this kind of a mapping in a resource constant scenario we consider a scenario where the resources of the architecture are limited that means say your grid dimensions is like this you have this grid dimensions and so you have SP's per SM earlier that we are consider was 40.

And now you are considering that your grid dimension 6, 2 just like earlier and block dimension was 5, 4 just like earlier. But now instead of considering SP's part SM as 40 let us start considering SP per SM is 20 so what happens now. Since you are SP's per SM is 20 at a time inside 1 SM I can execute only 1 thread block. So there will be further civilization for execution of thread blocks right.

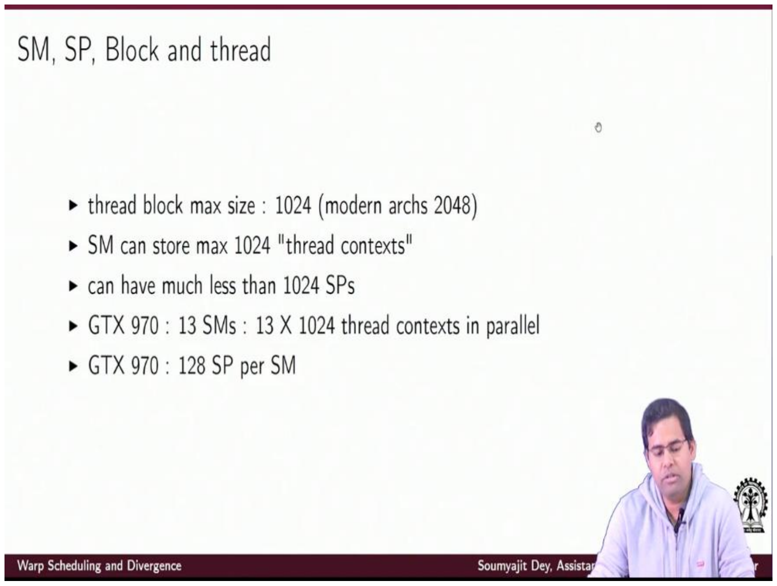
(Refer Slide Time 15:25)



So that would mean in 1 SM I have got only one block executing right. In another SM I have got another block executing. So with respect to the earlier example I have got more number of blocks waiting to execute because earlier I was executing more number of blocks per SM. So earlier I was executing with the 6 SM I was executing 12 blocks in parallel. But now with this 6 SM I am executing 6 blocks in parallel.

So the number of batches per thread block execution becomes large and its more sequential scenario right. So now I would have in this way less number of blocks executing parallel due to lack of hardware resources and more number of batching of thread blocks. So this batching of thread blocks and decision of which blocks goes to which SM would be differ will actually be decided by a global hardware scheduler resident inside the GPU.

(Refer Slide Time 16:21)



SM, SP, Block and thread

- thread block max size : 1024 (modern archs 2048)
- SM can store max 1024 "thread contexts"
- can have much less than 1024 SPs
- GTX 970 : 13 SMs : 13 X 1024 thread contexts in parallel
- GTX 970 : 128 SP per SM

Warp Scheduling and Divergence Soumyajit Dey, Assista

Now some general examples here like so when we speak of thread blocks SP’s and SM’s the maximum allowed thread block size is 1024 in modern architectures it can actually becomes 2048 it is also the property of the architecture. Now the reason is an SM can store maximum 1024 thread contexts. So every SM will be able to will as to it has to remember what are the threads that are map into it even if the thread is executing or not right.

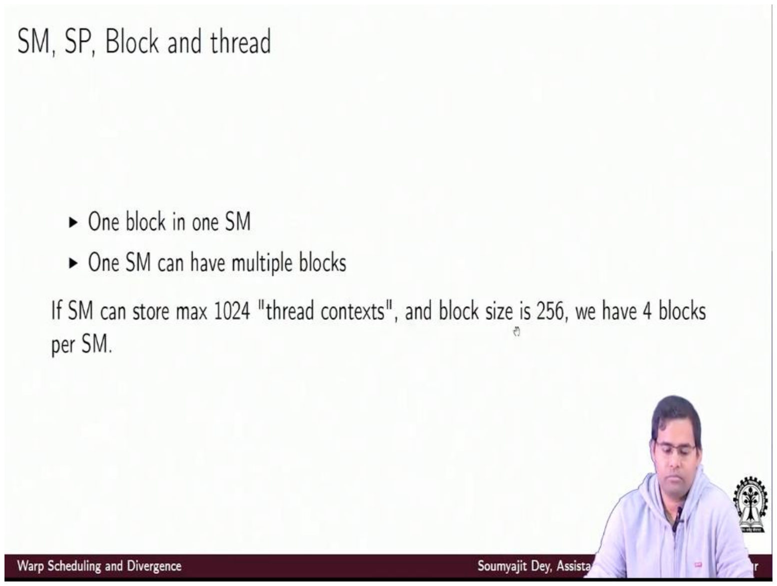
So that is the context of the thread. So that context needs to be stored and this limitation on the overall thread block size actually comes from the hardware requirement with respect to storing this maximum number of thread context so this tread context. So of course, the SM can have less than 1024 SP’s but it does not matter. As we discussed that I do not really need to execute all the threads in the block in parallel.

The constant of 1024 or 2048 for that matter comes from the SM ability to remember the context of the thread. It has been mapped a full block or may be multiple blocks. It should be able to remember the context of the threads. If the amount of memory available is something finite then only I mean this case it is 1024 thread context the amount of memory limited by that. Then only that many threads can be part of 1 block.

So although because even with that setting although the SM can have less than 1024 SP’s that means all the thread inside the block are not executing in parallel. But the SM can remember the context of the thread and actually schedule the threads using a smaller number of SP’s by remembering the context of the thread ok it executed up to this point then some other thread then other some other thread like that we will see some examples.

If we look at an example of GTX 970 then there are 13 SM’s. So overall it can actually remember 13 cross 1024 thread context in parallel and however it does not mean that many SP’s are available. It has got only one 128 per SM but it can remember the every SM can stored this many the number of thread context in parallel and accordingly it can schedule the execution of this maximum 1024 threads per block inside the SM’s by suitably choosing the threads inside the blocks and packing them together for execution on the SPs. So there is a there is where the scheduler in width which is inside the SM will come into play.

(Refer Slide Time 19:19)



SM, SP, Block and thread

- One block in one SM
- One SM can have multiple blocks

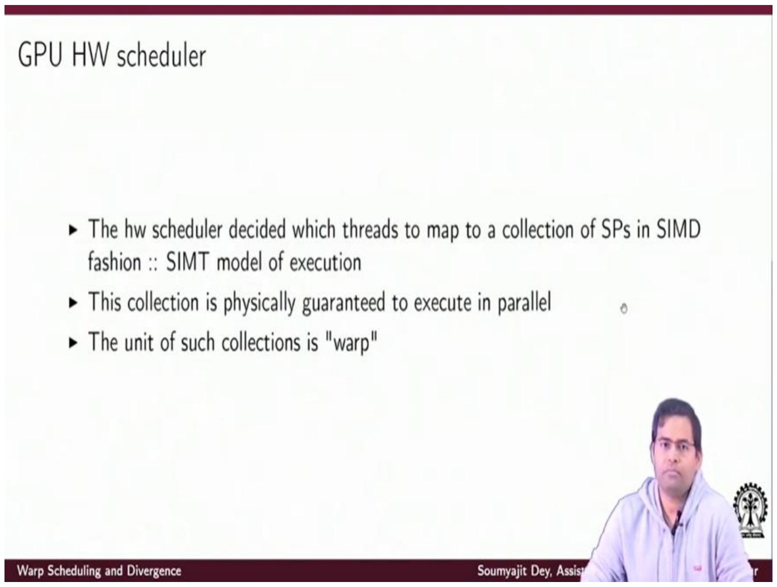
If SM can store max 1024 "thread contexts", and block size is 256, we have 4 blocks per SM.

So 1 block goes in 1 SM one block cannot be divided across 2 SM or more SM 1 SM can have multiple blocks. If can SM store maximum 1024 thread context and block size is 256, we have 4 blocks per SM. So 1024 is giving you the maximum block size that is allowed and of course you are free to actually define a block size that is less than that. But since the SM can store that max 1024 thread contexts and if you give a blocks size of smaller size let say 256.

Then when you are going to launch your kernel the SM can be mapped with 4 thread blocks. Your block size is 256 this is the important. Your block size suppose you have chosen to write a program where you have launched a kernel with block size 256. Since the SM can store maximum 1024 thread context that means you can remember the context of the this many executing threads while executing a sub part of they being parallel as define by the number of SPs.

But still since the max is 1024 your block size is smaller value of 256. The SM can be mapped with 4 thread blocks 256 times 4 is 1024 and this mapping is decided by the high levels scheduler who is distributing the blocks across the SMs. So the number of blocks that will be mapped to the SM is defined by the number of threads per block while remembering the total number of thread context then the SM can be remember.

(Refer Slide Time 20:56)



GPU HW scheduler

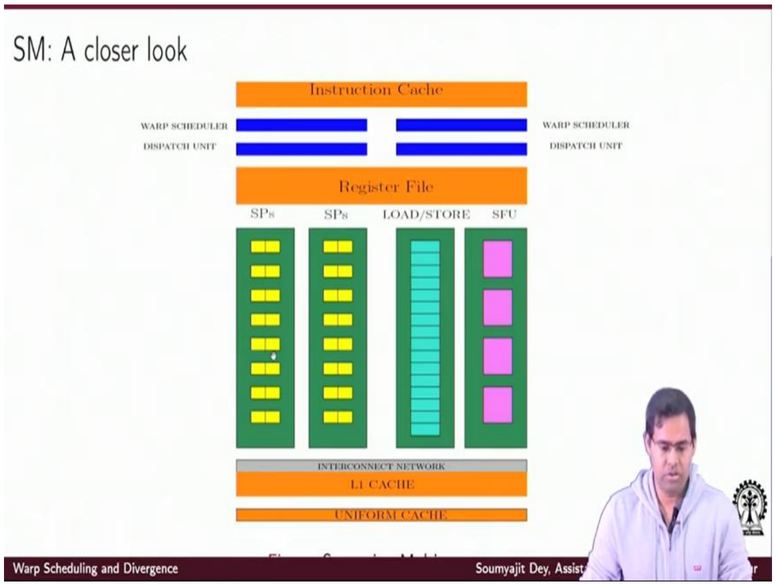
- The hw scheduler decided which threads to map to a collection of SPs in SIMD fashion :: SIMT model of execution
- This collection is physically guaranteed to execute in parallel
- The unit of such collections is "warp"

Warp Scheduling and Divergence
Soumyajit Dey, Assis

So this is what the hardware scheduler decides. It decides which thread block will map to a collection of SP’s or the or to map SM and inside the SM we have a secondary this hardware scheduler which decides that ok this are the thread blocks that has been mapped to this SM. Out of this thread blocks which are the physical threads which would be packed together for an execution. This binding of physical threads together for a lockstep execution is what we have defined earlier as a warp.

So this is the basic unit of execution inside a GPUs SM. So this is done by the hardware scheduler blocks sitting inside the SM. So the SM will be assigned a set of thread blocks or maybe one thread blocks. Inside the threads blocks you have multiple of threads which of the thread will actually execute in parallel in units of the execution that is warp will be decided by the GPUs SM hardware scheduler.

(Refer Slide Time 21:56)

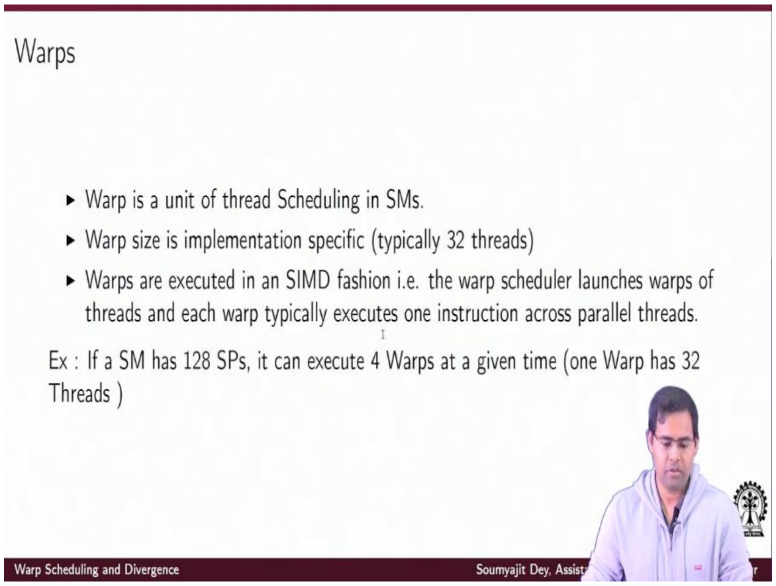


SM: A closer look

Warp Scheduling and Divergence Soumyajit Dey, Assist

So this is basically a way we are trying to give the picture. So we have an SM inside the SM you have this warp scheduler as a hardware unit is deciding which of the threads are going to be dispatched together through the SP's for execution into execution in lockstep.

(Refer Slide Time 22:17)



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)

Warp Scheduling and Divergence Soumyajit Dey, Assist

So just to summarize the warp is the unit of the thread scheduling in the inside the SM's. Warps size is implementation specific typically it is still now 32 threads inside a warp. Now this warp is executed in the SIMD fashion that is the warp scheduler launches a warp of threads each warp typically executes one instruction across parallel threads. If a SM has 128 SP's it can execute 4 warps at any given time. One warp has 32 threads right.

So a question is what is a part of a warp which are the thread id that are the part of the same warp that is what decided by the warp scheduler sitting inside the SM. So just to review it here the warp scheduler decides which thread to pack together for execution in a warp. These are all threads with consecutive thread ids but so the threads in the warp are guaranteed to the execution the same SID SIMD distribution instruction together.

It executes one SIMD instruction followed by the next SIMD instruction in lockstep like that. Different warps can actually progress to the SM together. Why? Because I have more than 32 SPs. For example I have let say 128 SP's. So in parallel 4 warps may execute together. So inside a thread block I have lot of threads. This thread are packed into packets of 32 by the warp scheduler.

This packets of 32 threads are warp will execute into lockstep together from the programmers points of view in terms of a instruction right. But again the warps may progress a different speeds. Each warp progress each warp ensures that the threads inside the warp progress at the same speed. But I have different multiple warps can actually going here which warp goes in here that is decided by the warp scheduler.

So which are the real warps that are executing it depends. That is why when I launch a thread block containing that number of threads as we have discussed earlier there is no guarantee that all the threads are progressing at the same speed. The threads will be packed into warps as decided by the warps scheduler. And it will decide which packet of thread will progress at which speed that is why we will require the SIM thread or some kind of some synchronization primitive like that to ensure or force synchronization among threads.

So that they actually have a consistent view of data points on which they want to collaborate and work together. With that we would like to end this lecture and we will resume from this point thank you.