GPU Architectures and Programming

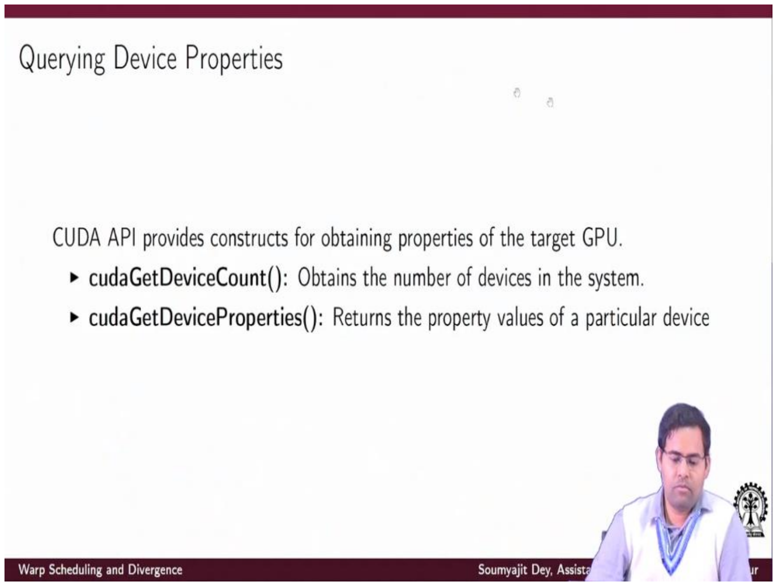
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 04
Lecture No # 18
Warp Scheduling and Divergence (Contd.)

Hi welcome to the lecture series on a GPU programming and architecture. So in the last lecture we have been discussing about how to query the GPU device properties and there is a good reason behind that you can write your program in a architecture oriented parameterized manner. So that you can first run this kind of diagnostic code to figure out what are the different architecture parameters.

And accordingly you can decide the launch parameters and some other parameters of your kernel and so that it executes optimally in a given architecture with respect to the execution time of the kernel.

(Refer slide Time 01:07)



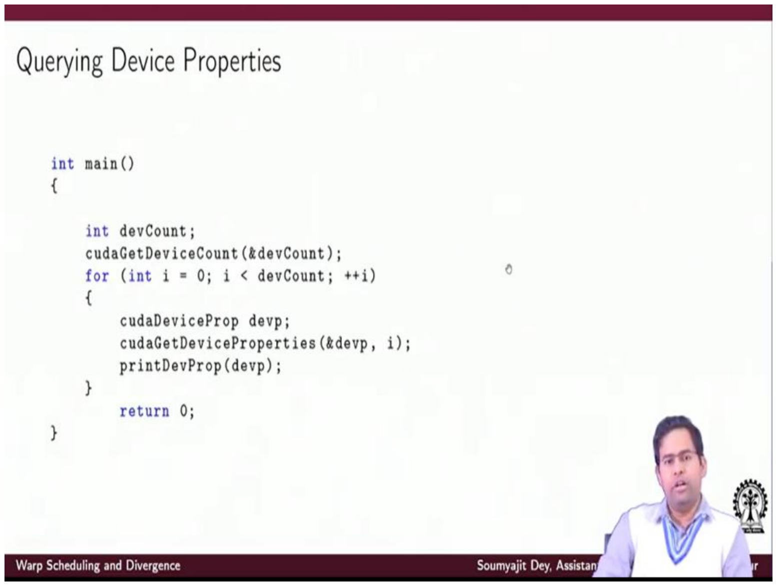
Querying Device Properties

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

So we were discussing some specific CUDA APIs for target GPU systems. For example if I use this CUDA Get Device Count function call I get to know what is the number of device in the system. I can use a CUDA Get Device Properties called to return different property values for a particular device.

(Refer Slide Time 01:23)



Querying Device Properties

```
int main()
{

    int devCount;
    cudaGetDeviceCount(&devCount);
    for (int i = 0; i < devCount; ++i)
    {
        cudaDeviceProp devp;
        cudaGetDeviceProperties(&devp, i);
        printDevProp(devp);
    }
        return 0;
}
```

Warp Scheduling and Divergence

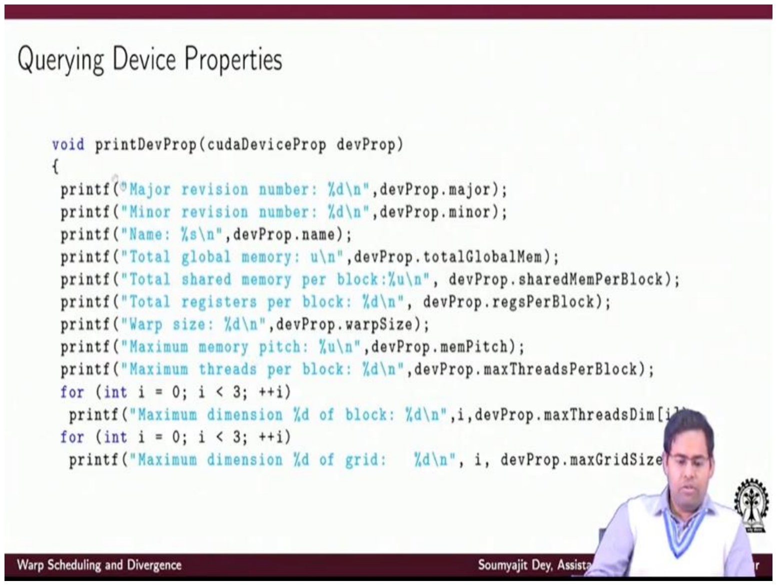
Soumyajit Dey, Assistan

Now just we will see some highlight of an example program. So consider this scenario that you have a small host side code from which you are launching this CUDA Get Device Count function call and it would be returning the get device count in the variable that has been passed here right. And then you are going to use this initialize variable the variable get initialize from this call.

You are going to use it to upper bound the number of iterations for the subsiding loop and then in using this loop the first thing you do is you initialize type a variable devp of type CUDA Device Property and then for each index value i of this loop from i equal to 0 up to the device count you are making queries part device right. So you are now calling the function CUDA Get Device Properties passing it structure of type CUDA Device Prop the name of it is devp her in this code and passing it the device id. Device id 0, device id 1 up to the device count the total number of devices we do a of course to the last device.

So in that way for every device id you have the corresponding device information getting stored in the CUDA in the structure devp of type CUDA Device Prop and there that again you are going to print using this specific function print device property right.

(Refer Slide Time 03:03)

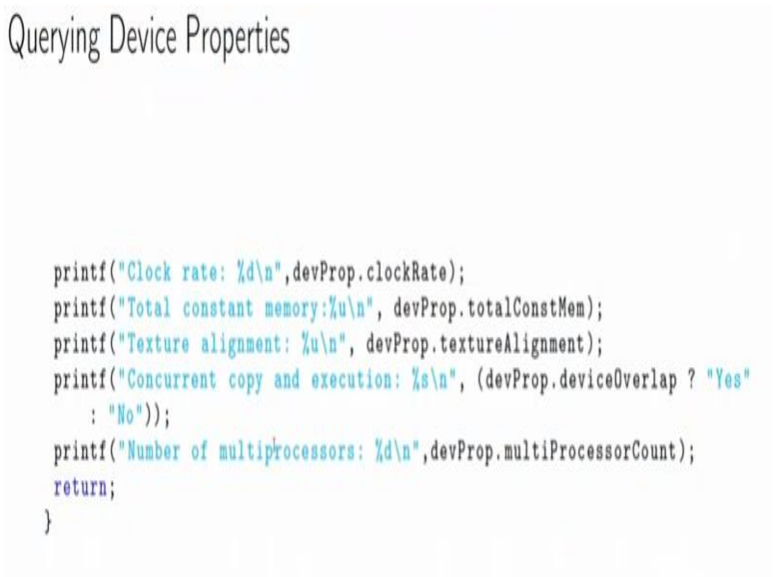

Querying Device Properties

```
void printDevProp(cudaDeviceProp devProp)
{
 printf("Major revision number: %d\n",devProp.major);
 printf("Minor revision number: %d\n",devProp.minor);
 printf("Name: %s\n",devProp.name);
 printf("Total global memory: u\n",devProp.totalGlobalMem);
 printf("Total shared memory per block:%u\n", devProp.sharedMemPerBlock);
 printf("Total registers per block: %d\n", devProp.regsPerBlock);
 printf("Warp size: %d\n",devProp.warpSize);
 printf("Maximum memory pitch: %u\n",devProp.memPitch);
 printf("Maximum threads per block: %d\n",devProp.maxThreadsPerBlock);
 for (int i = 0; i < 3; ++i)
  printf("Maximum dimension %d of block: %d\n",i,devProp.maxThreadsDim[i
 for (int i = 0; i < 3; ++i)
  printf("Maximum dimension %d of grid:   %d\n", i, devProp.maxGridSize
```

Warp Scheduling and Divergence   Soumyajit Dey, Assista

So this is something which you have customized for our purpose the print device property function. So essentially it is being passed this device properties structure right. And so essentially in this case this it is it is expecting that it would be given a CUDA device property types structure and that it would be having several other fields which will be printing through this print f statement. So I am not going to discuss this print f statements in details does not make much sense.

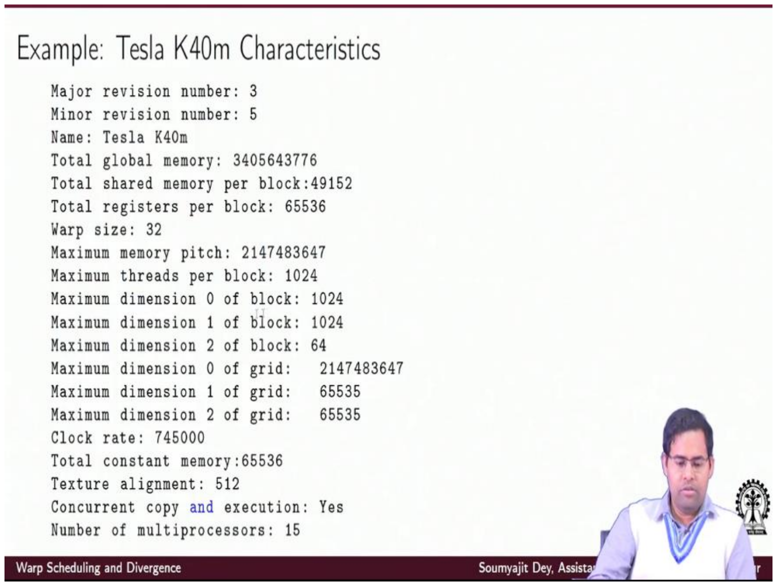
(Refer Slide Time 03:36)


Querying Device Properties

```
printf("Clock rate: %d\n",devProp.clockRate);
printf("Total constant memory:%u\n", devProp.totalConstMem);
printf("Texture alignment: %u\n", devProp.textureAlignment);
printf("Concurrent copy and execution: %s\n", (devProp.deviceOverlap ? "Yes"
    : "No"));
printf("Number of multiprocessors: %d\n",devProp.multiProcessorCount);
return;
}
```

Rather than that let us look at how the code I mean what really the code will output and of course the slides will be with you and you can have a look into how is essentially different print f statements are organized.

(Refer Slide Time 03:49)



Example: Tesla K40m Characteristics

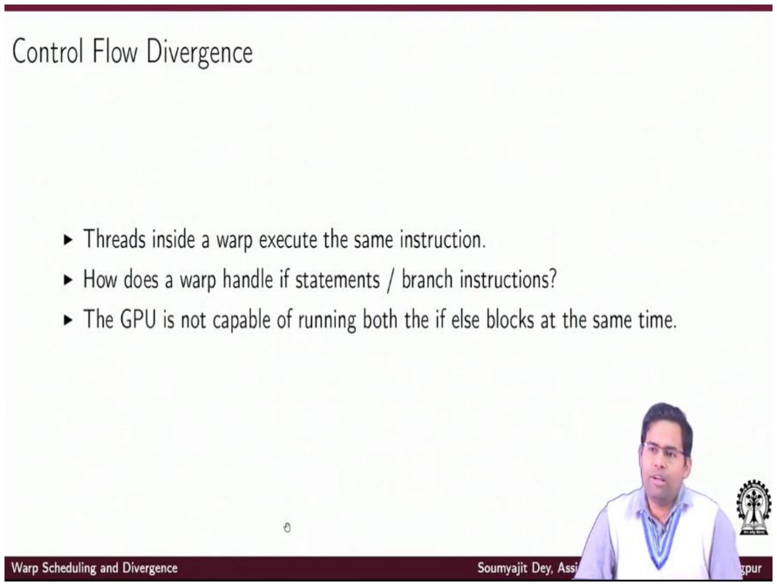
```
Major revision number: 3
Minor revision number: 5
Name: Tesla K40m
Total global memory: 3405643776
Total shared memory per block:49152
Total registers per block: 65536
Warp size: 32
Maximum memory pitch: 2147483647
Maximum threads per block: 1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate: 745000
Total constant memory:65536
Texture alignment: 512
Concurrent copy and execution: Yes
Number of multiprocessors: 15
```

And so overall if I run that code these are the different device properties I can extract. So I can let I can actually extract the different revision numbers of the architecture. So that is basically the device properties major and minor field. So this major and minor field is giving telling me what is the major revision number and the minor revision number of the architecture the name of the architecture family. So that is the name of the device family and then I have the total global memory size available for each device.

So for this device I have a total memory size like this and then there are several other memory segments for example these are total global memory. What is the amount of shared memory per block? What is the amount of registers per block? What is the warp size maximum memory pitch? What is the maximum number of threads allowed per block? And for each of the block definitions what is the maximum dimension of the block in each of the 3 possible dimensions.

And finally lifting from blocks to grids what is the allowed dimension of the grid in the 3 possible dimensions x y and z. So these are the possible value that get printed and of course the GPUs clock rate. The total amount of constant memory that is there the texture memory and whether the GPU support certain other properties like concurrent execution of kernel. What are the total number of SM in the GPU and all that? So if we execute the earlier diagnostic code these are the different messages that can get printed and they can be useful for framing your launch parameters of the kernel.

(Refer Slide Time 05:42)



Control Flow Divergence

- Threads inside a warp execute the same instruction.
- How does a warp handle if statements / branch instructions?
- The GPU is not capable of running both the if else blocks at the same time.

Warp Scheduling and Divergence Soumyajit Dey, Assi gpur

Now so that is about how you can extract the device properties now since we are actually discussing about warps scheduling the warps there is a another major issue which impedes the performance of a kernel which is called control flow divergence. Till now we have been just discussing that when you are launch the kernel you are essentially launching multiple thread blocks. Each of the thread blocks progressed at its own speed. Inside a thread block I have a collection of warps that are decided by the internal schedulers and the warps will also proceed at their own speed. Since each of the essence able to execute a lot of warps together in parallel.

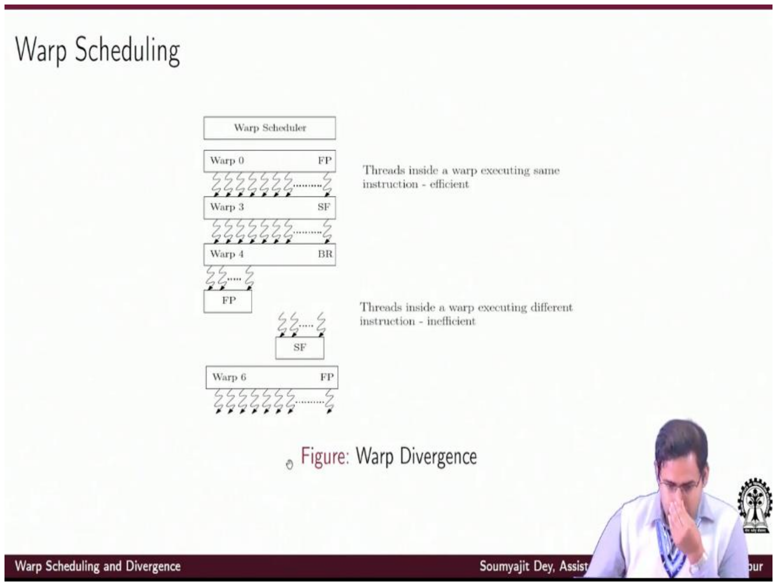
It can tolerate the latencies of long latency operations and in that way a GPU is able to make progress with lot of threads in parallel. The progress of each thread may be slow due to latency of operations. But overall the GPU would be giving you high throughput because it is completing a lot of operation in parallel. However, this question that how many operations really the GPU executing in parallel depends on something very important which is the flow of control inside the kernel.

We have been assuming the threads can make progress in parallel inside the warp but that really did not always be the case for example we know that by definition of a warp that the thread inside the warp execute the same instruction. At any point of time they are executing from a programmer’s point of view they are executing the same instruction. What if there is a branch statement which the warp is executing and some of the threads in the warp satisfying the condition of the branch where some of the threads in the block do not satisfy the condition of the branch.

The GPU is not capable of running both the if and else blocks at the same time. That is not how it is designed that it will make progress with the warp or it will form the warp in to 2 parallel lines. So that is some of the threads progress with the if part of the code and some of the threads in the same warp would progress with the else part of the code. Speaking I mean if you just think logically that also does not make sense because if you allow the hardware to fork a war and still progress in parallel there can be further subsequent forks right.

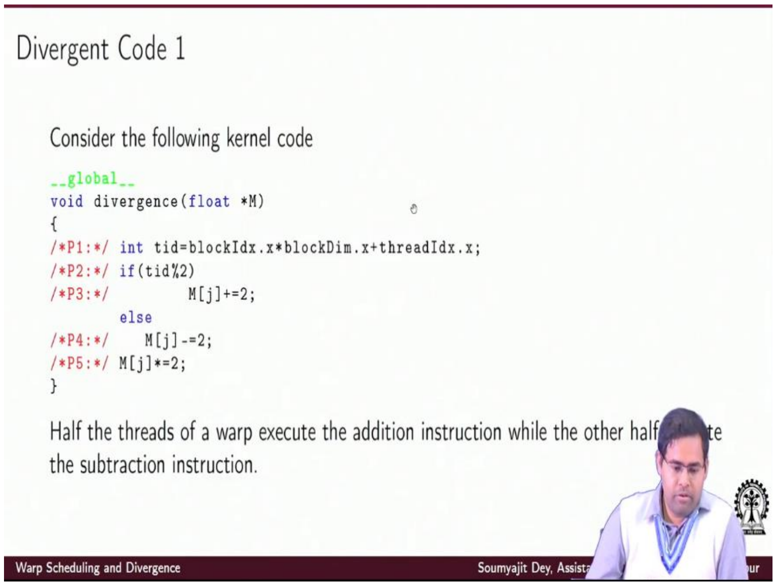
So how will they really hardware able to handle so many numbers of forks of a warp and make it progress does not make sense in terms of implementation complexity right. So we are not getting into that detail right now. But rather we will try to understand how really a GPU handles such divergence of control flow that comes from branch executions.

(Refer Slide Time 08:32)


Warp Scheduling
Warp Scheduler
Warp 0 FP
Warp 3 SF
Warp 4 BR
FP
SF
Warp 6 FP
Threads inside a warp executing same instruction - efficient
Threads inside a warp executing different instruction - inefficient
Figure: Warp Divergence
Warp Scheduling and Divergence
Soumyajit Dey, Assist

So suppose you have a warp scheduler and it is executing these warps and as long as the thread inside the warp are executing the same instruction it is a very efficient situation. The problem comes as we discussed that threads inside a warp start executing different instructions. So then the number of threads which are executing in parallel will reduce and we will have less amount of throughput and that required that is phenomenon which is known as warp divergence which leads to performance loss from the perspective of the GPUs throughput.

(Refer Slide Time 09:12)



Divergent Code 1

Consider the following kernel code

```
__global__
void divergence(float *M)
{
/*P1:*/ int tid=blockIdx.x*blockDim.x+threadIdx.x;
/*P2:*/ if(tid%2)
/*P3:*/         M[j]+=2;
        else
/*P4:*/    M[j]-=2;
/*P5:*/ M[j]*=2;
}
```

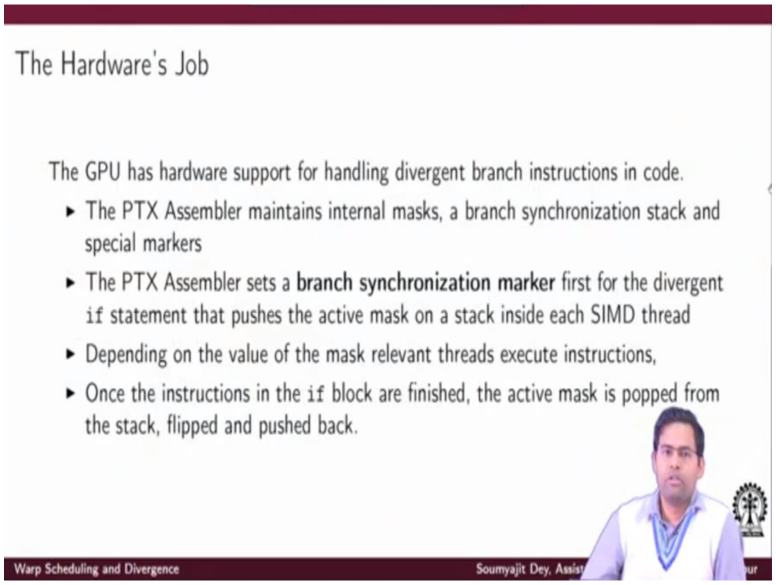
Half the threads of a warp execute the addition instruction while the other half [illegible] te the subtraction instruction.

So as an example let us consider the divergent code. So this is the kernel where it is a very simple kernel we are just labeling each of the statements in this kernel with levels P1, P2, P3, P4, P5 there is a reason for that. So the program statement with label P1 just computes the global thread id for that specific kernel for that specific thread in the kernel. The program statement with level P2 decides whether that thread will actually satisfy or not satisfy the if block.

So essentially all the thread id which are even would satisfy it and they would get in to do the execution of the statement P3. The thread ideas which are odd indices would actually get in and execute the statement P4. And after that both all the threads together would execute the statement P5. So overall here we have the scenario that since the if condition is tid percentile 2 half of the threads of a warp would execute the addition instruction while the other half of the threads in the same warp would execute the subtraction instruction.

And as we have decided it is not the case that the warp would actually split, and half of the instruction execute one instruction and half of the threads execute the other instruction in parallel. That is never going to happen in the GPU.

(Refer Slide Time 10:38)



The Hardware's Job

The GPU has hardware support for handling divergent branch instructions in code.

- The PTX Assembler maintains internal masks, a branch synchronization stack and special markers
- The PTX Assembler sets a **branch synchronization marker** first for the divergent `if` statement that pushes the active mask on a stack inside each SIMD thread
- Depending on the value of the mask relevant threads execute instructions,
- Once the instructions in the `if` block are finished, the active mask is popped from the stack, flipped and pushed back.

Warp Scheduling and Divergence

Soumyajit Dey, Assist

So how really does the hardware handle this scenario? So the GPU does the following whenever such a branch comes. So in if we look at the corresponding PTX code for this kind of kernel we would see that the PTX assembler maintains internal masks or essentially the execution of all such instructions is what we call as predicated instructions. So it maintains the mask it is a bit pattern which would decide which instruction would really be executed and it and the status of the instruction are actually store in something called a branch synchronization step.

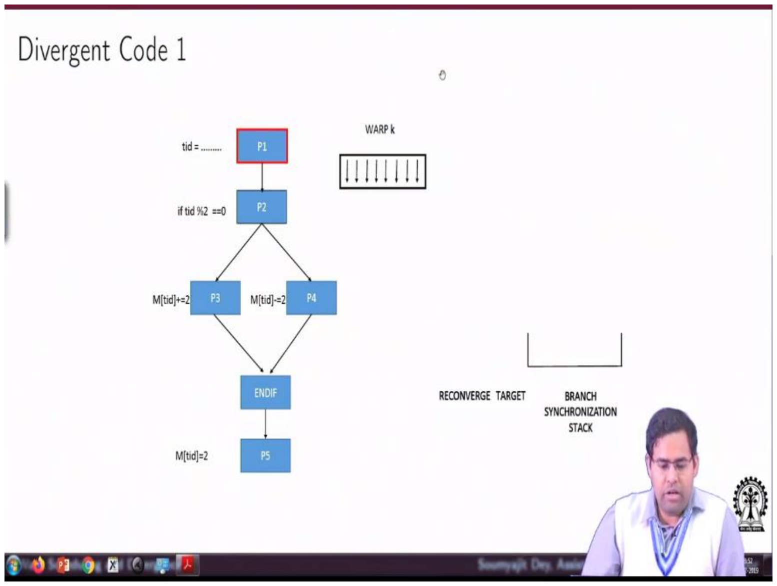
So the PTX assembler sets the branch in synchronization marker first for the divergent statement the if statement and pushes the mask on the branch synchronization step for each of the SIMD thread. It will decide that whether the mask value would actually decide which thread is executing or not executing. Depending on the value of this mask the warp scheduler simply makes progress with the instruction.

And once the instructions for the eve block are finished then the active mask is popped out from the stack and it is flocked. So when it is getting flipped essential the mask changes. So which ever where the one entries will now change to 0 and whichever were the earlier 0 entries would now get changed to 1. So essentially the other instructions the other thread ids which we are now stalled would now start to executing essentially then the once the if block finishes the else block starts execution.

Well it is too verbose a description possible so we would make it much more alive through a running example in this case. Just a brief summary in this case would be that when the corresponding assembly code is generated a suitable marker is decided right. And at the run time when this predicated instruction, instructions with markers are getting processed a separate data structure a branch synchronization stack is set up.

And which instruction would make progress and which instruction would not make progress is actually encoded as a 1 0 pattern and stored in the branch instruction stack. So when the if block progresses the threads corresponding to (()) (13:00) I have a 1 in that stack would make progress and once the if block ends I actually bring out this 1 0 pattern flip it and if I flip the pattern now essential then I have in terms of the hardware it would know that what ever is corresponding to one in the pattern would actually be the threads in the else block and now they would make progress.

(Refer Slide Time 13:28)



Let us look at it rigorously from with a example here so this is what we called a control flow graph of the corresponding program we had earlier. So we are making good use of the levels here. So just have a re look into the different levels P1, P2, P3, P4, and P5 for the kernel. So P1 is for the thread id computation, P2 for if, P3 for the addition then after the else I have the P4 for the subtraction operation and finally the common operation has got P5.

So here I have P1 followed by P2 the branch that is a if condition if tid percent 2 is 0 and then I have a fork here to P3 or P4 so some threads would go in here and some threads would go in here and once this if condition ends I have the thread synchronization together and then I have all them executing P5 that is this operation. So I believe this actually should be star equal to 2 here right.

So now let us try and find out that how the warp progresses with the help of this kind of a branch synchronization step and the predicated execution of the masks that we have been discussing earlier. So this is how let us say these are the threads which are going to execute for the kth warp and they are right now in executing the instruction for the level P1. There they are computing all of them computes the P thread id.

Then all of them comes to the branch instruction so when all of them comes to the branch instruction it will actually see that the branch condition is satisfied by the threads with even ids right. So those who actually satisfy would be considered as 1 and the thread id which do not satisfy is consider as 0 and this way active mask is created the 1 0 pattern, and this initialized active mask is now stored into the branch synchronization step.

Essentially the location of the 1 which decide which threads make progress and this is how the hardware takes care of making progress with the warp. It simply looks into the active mask accordingly the scene as we discussed that this actually signified which of the predicated instruction would make progress. Since each of the even thread ids are having 1. So those are the thread ids that would progress as you can see that now the I have half of the threads making progress that the tids which are even.

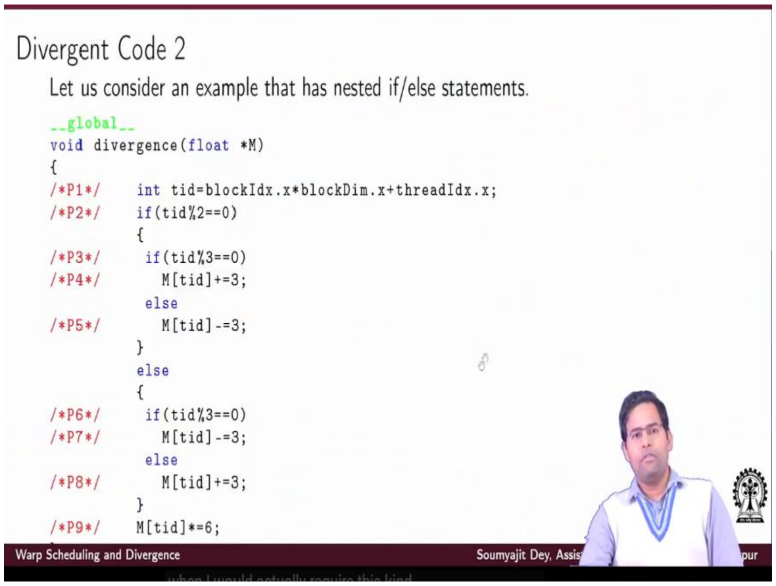
So they would all execute P3 once that is done once the execution of P3 is done then this mask bits are complemented. So P3 is done I have complementation of mass bits so this pattern is going to change the reverse pattern which is this with the reverse pattern the hardware will again start executing the mask instructions following this mask. So it should essentially use this sequence to compute which of the threads will make progress.

These are the alternate threads which are making progress after this all the threads are suppose to converge together and start executing together. So that would mean from the branch instruction branch synchronization step the mask is pop and since there is nothing here so the hardware warp scheduler would simply make progress with the entire warp right. So the question which is obvious here is why do I at all need a stack kind of thing because here I was simply making progress with the warp.

Since there was a branch instruction I actually computed which are the threads that satisfy the branch from a hardware implementation perspective I represented that with a 1 0 bit pattern which ever has got 1 which tid was 1 corresponding to that I said some mask some the predicated instruction has active and I made them flow corresponding threads miss progress. And after those threads executed P3 I just in the in terms of the hardware I just reverse the bit pattern.

So then I could get the other thread ids to make progress with the alternate flow of instructions or alternate flow of instruction which was in P4 and finally I just well I I actually reached the synchronization point and at that point I just set of the mask and I just made progress with all the threads right. So that is the simple thing consider the scenario where you have this kind of cascaded ids which is so common in a general program how would you then handle it? For that is when I would actually required this kind of branch synchronization step.

(Refer Slide Time 18:37)



Divergent Code 2

Let us consider an example that has nested if/else statements.

```
__global__
void divergence(float *M)
{
/*P1*/    int tid=blockIdx.x*blockDim.x+threadIdx.x;
/*P2*/    if(tid%2==0)
          {
/*P3*/     if(tid%3==0)
/*P4*/       M[tid]+=3;
           else
/*P5*/       M[tid]-=3;
          }
          else
          {
/*P6*/     if(tid%3==0)
/*P7*/       M[tid]-=3;
           else
/*P8*/       M[tid]+=3;
          }
/*P9*/    M[tid]*=6;
```

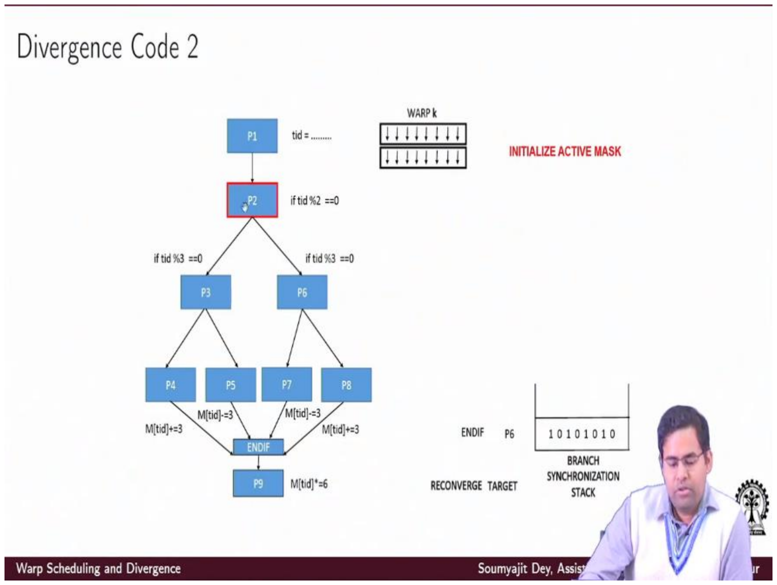
Warp Scheduling and Divergence   Soumyajit Dey, Assis

So we consider this second example of a divergent code now this is where we are actually considering nested if else statement and we are trying to figure out how this nested if else statements would actually be handled by our divergence handling scheme in terms of that that branch synchronization stack. So just look at into it very minutely first we compute the tid, if the tid is even we get into a branch inside this branch we also check whether the even tid are divisible by 3.

So only those tid which are overall dividable by 6 would get into this part and do an addition by 3. Otherwise the event tids which are not divisible by 3 would get into the else block and subtract 3. For all the odd tid if they are divisible by 3 you have something if they are not divisible by 3 you have something else and finally everything converges to location P9. So this is like more complex program the complexity is just in terms of the nesting of if else statements.

Have a relook into the original program we had a single level of if else is. Here in this program we are considering for the nesting of if else statements.

(Refer Slide Time 19:58)



Look in to the control flow graph here so you have positions P1 for tid computation, P2 for the first branch, P3 for the next branch which is only satisfied for everything which is percentile 2, 0 and also percentile 3, 0 that means tid which are divisible by 6. Then you execute P4 otherwise you execute P5. When this both are done you execute P6 then you execute P7 and then P8 finally everything would synchronize and you go to P9 right.

So this is the structure of the control flow graph. Now let us look into our original idea of warp execution. So your warp starts you initialize the active mask so at P2 you compute the mask which is basically the sequence of 1 0 1 0 like that because tid percentile 2 is 0. So all the even thread ids are satisfying the mask. So you make progress here with all the even tids and you are here.

And then what you do is again since you are facing a nested if so this is the situation which we were discussing that you have you are you have already taken care of first if based on that if you are pushed in a mask. Once inside the if part of the logic you are facing the second if right. So for this second if you have this condition so the first you do is you push the old mask into the step. What is the old mask? Therefore, the old mask simply signifies what is your original thread of computation?

The old mask signifies execution of the if part of the first nest right. Now you are actually going to come into the if part of the second nesting. So how to do that so you have pushed the old mask once again and then you are applying this condition on the old mask. So essentially you are trying to figure out which of the tid's are going to progress and execute P4 right. So figuring that out would mean applying this condition of the if on the old mask. So and finding out which are the threads are really going to execute here.

Now as you can see from the condition only the threads which are divisible by 6 in terms of tid will execute P4 and that is what it is. So it is the old mask and you apply the condition here so that would give you all 0 and only 1 at the 6 position of the tid 0, 1, 2, 3, 4, 5, 6 only a 1, 1. So only this thread id will proceed an execute P4 that is all is happening. After that you are supposed to execute P5 how will that happen.

So all you need to do is you have to compliment the element bits of this entry in the step right. So you just compliment the entries because this was your old mask this is your old mask 2 instances. Second instance because of the nesting first you apply your mask on the instance and figure out how to execute who will execute the second if and then you just reverse this pattern with the reverse pattern you execute the else inside the if.

So once this is done so this is the progress of this 3 threads here. So once this is done you will just pop the step why because this segment of P3, P4, P5 execution is done. So you are essentially done with the if part of the first level if. Now you are going to process the else part of the first level if. How do you do that? It now you are using the older logic. So all you are going to do is you are going to compliment the relevant bits. So this this was your original bit pattern on the old mass you complement the relevant bits.

Once you complement the relevant bits you have the other half of the warps progressing to P6 where again you have another mask to execute. So again follow the old algorithm you push the old mask so again this is the old mask corresponding to the reverse pattern here. So that you are taking care of the other threads who are executing inside this else. So for this old mask again you will apply this condition tid percentage 3, 0.

If you apply this condition you get only this thread which will execute here why because it has to be an odd thread id 0, 1, 2, 3 but still it has to be percentile 3 equal to 0. So it will only be made by thread id 3 because the next thing that is thread id percentile 3 is 0 is 6 which is not odd. So it would definitely be a 0 here. So you will have only one thread progressing and then the rest of the threads needs to progress with the else part for that again you have to compliment this mask.

So this mask is complimented and according to the hardware decides which other threads would progress. Once this is done so you have got all the diverging threads re converging back here in this ENDIF and that is signified by popping the step. So after this there would be another synchronization. So first time you pop the stack to identify that ok this sequence of second level if have converged.

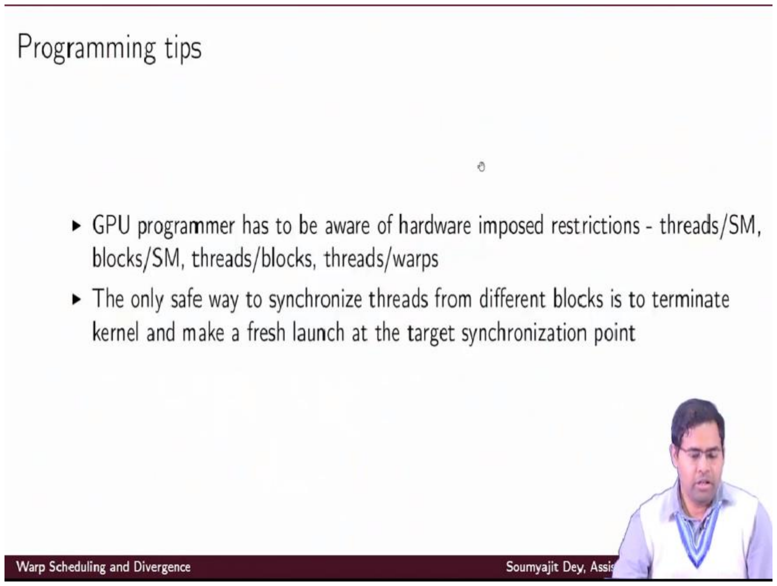
But then you figure that well this the first level if has also converge right. So that is signified by popping the stack again. I hope this is clear there were two pops the first pop signifying that ok this flow. Let me mark the flows here so the first pop signifies here that this part of the flow has converged and then you will pop the step again and empty it because that would signify that the from the high level the entire flow has converged. So you pop the step completely.

So overall this is the idea that in terms of the hardware and the PTX predicated instructions. How GPU would really handle this divergent of warps and then re converging back the threads in the warp. One important thing you notice here is that whenever the warp diverges you have performance loss. Why because whenever the warp diverges you have less number of threads executing per warp because whenever I have a divergence half of the threads are not half.

The threads which satisfy the if condition would progress and then subsequently at some other point of time the threads that is satisfying the else condition is progress. So essentially I have the warp making progress with some of the threads executing which satisfied (()) (27:21). And then at some point of time some other point other threads executing which satisfies the else. So whenever there is a divergence in your execution of a CUDA program you have performance loss.

So divergence is not a good behavior for your program and there can be there are a lot of research which go on in developing tools and technique which identify whether you whether your program can exhibit divergence how whether it may be possible to over come it by an alternate program and things like that. But a divergence is a serious issue which as a programmer you need to be aware of. So that you can understand the way the program is going to behave in terms of architectural timing.

(Refer Slide time 28:13)



Programming tips

- GPU programmer has to be aware of hardware imposed restrictions - threads/SM, blocks/SM, threads/blocks, threads/warps
- The only safe way to synchronize threads from different blocks is to terminate kernel and make a fresh launch at the target synchronization point

Warp Scheduling and Divergence   Soumyajit Dey, Assis

In this lecture we would like to summarize with some specific programming tip that as a GPU programmer you have to be architecture aware that means you need to be aware of the hardware imposed instructions the hardware impose restriction in terms of how many threads can be mapped into an SM.

How many blocks can be mapped into an SM? How many threads are allowed per block and how many threads exit execute inside a warp? things like this because these are the figures which actually help you to write a performance in our program. And something are very important with respect to our earlier coverage of synchronization of threads it would be this. As we have discussed that synchronization primitives of GPU programs work inside thread block right.

So suppose you are trying to synchronize or have collaborative execution among threads across multiple blocks that is not allowed. So the only safe way to synchronize threads from different blocks would be to terminate the kernel right at that point that would mean that all the computation get done at that point and then you will make a fresh launch of a kernel at that point. so that all the threads would get launched again with a consistent view of the memory. So with this we will be ending this lecture thank you.