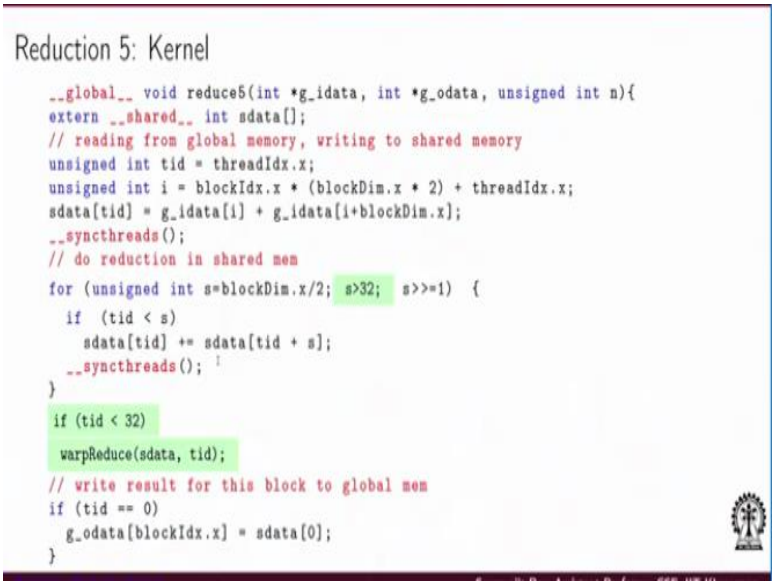
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology - Kharagpur

Lecture – 30
Optimising Reduction Kernels (Contd.)

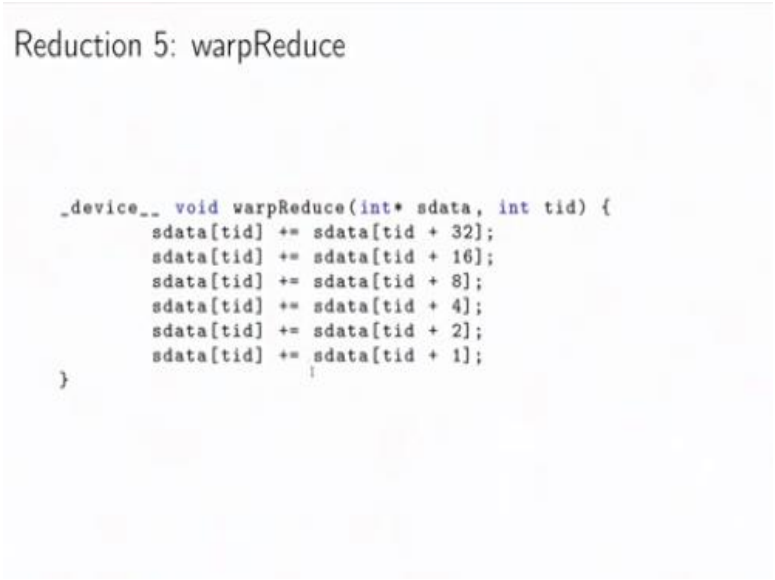
Hi, welcome back to the lectures on GPU architectures and programming so, in the last lecture we have been discussing some more parallel reduction techniques and so, if you remember, there were 2 specific optimizations we talked about in the last lecture oh, sorry, 3 specific optimizations; one was the unrolling of the last warp and after that we also discussed unrolling the entire computation.

(Refer Slide Time: 00:57)



And at the end, there was algorithmic cascading right, so before moving on we will just revisit here for some more intricate points for each of these 3 optimizations that we had. So, first of all when we were trying to do the unrolling for the last warp, so this was the code here. So, when the tid is less than 32, so for that we are just unrolling the last warp and calling this to warp reduce function.

(Refer Slide Time: 01:29)



Now, for the other tid's, we will be executing the normal loop and just to be sure, I want to check out here that why certain things went missing in this code, we actually discussed what is the advantage of writing it like this. First of all if you remember, that we could remove the loop out of the equation and that actually remove the loop overhead associated and essentially, the loop behaviour of the last warp gets executed in sequence.

So, I do not have the checks of the loops to be done again and again, right because as we can say this is completely unnecessary because I am essentially writing the same behaviour that would repeat for all the statements here and since the loop is not there, I do not have to do this check by executing unnecessary instructions; advantage one. The second advantage was that since, is the warp of instructions so, I do not need the same thread as we have already discussed.

And also something important that we have here that there was this check inside the loop whether tid is less than s or not right. Now, of course when I am unloading the loop, these are the instructions which I should repeat but as you can see that if instruction is also missing. Now, I feel it is very easy here to understand that we do not really need the if instruction here because it says that okay, if I am only interested in the threads inside the stride here.

But here, since I am only considering the last warp and going to consider the warp with the thread id is from tid 0 to 31 right, so technically I only want that warp to make progress here so, the issue is why do I remove this if condition? Well, even if the other threads are executing, I do not mind because finally, this is the warp which I am interested in which

contains the thread id's from 0 to 31 so, only those will actually get through this if condition, right.

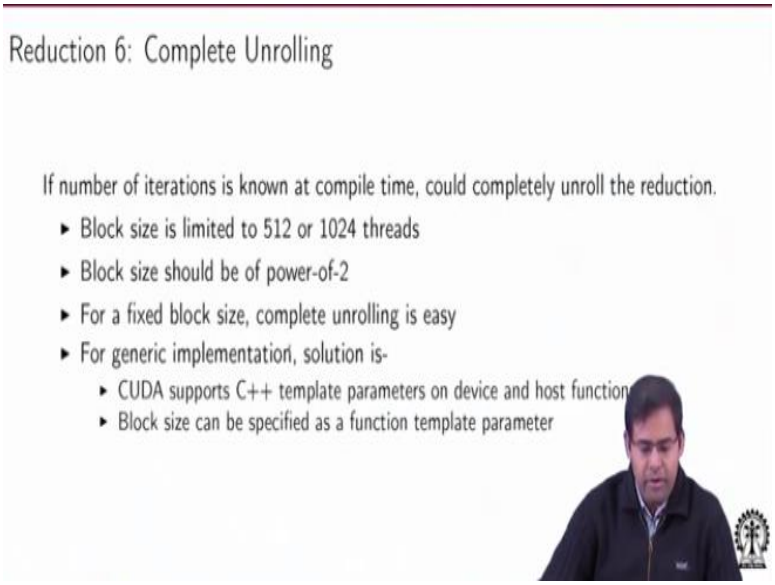
So, this if condition is already ensuring the warp that actually gets executed for this warp reduce function is basically, this is basically this sequence of instructions and here only the tid's that are having the value 0 to 31 will coming here as a complete warp, right. Now, since I have; I am actually interested to execute the warp and let it flow through all these sequence of instructions.

And I do not really care whether I have this check of tid less than s or not right, I mean it is easy to see here, you can just check that okay, finally I will just pluck out the result from the thread with tid 0 but since, the warp is anyway progressing in lockstep is completely unnecessary to put in that condition again for each of this instructions here. I do not; I hope that is clear now, that since here I only have this warp containing the instructions 0 to 13; the thread id is 0 to 31 executing together.

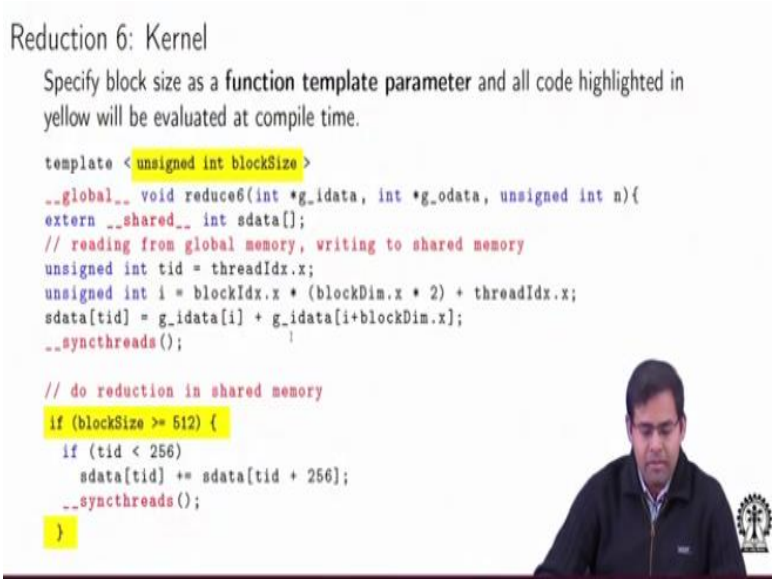
They are anyway executing in lockstep, if I put in a check here, it really does not matter right, only thing that will have is happen is that it will give me some more instruction over it, I want to eliminate that so, for that reason it does not; it is not present here and anyway, functionally I get the value computed that I required and it is stored in gID 0, right. So, of course all the tid's are doing the computation but by the behaviour of the code, I will get s data 0 containing the final value once all the threads inside this sequence finish executing their respective instructions here, fine.

So, that would be our discussion for the warp produced just wanted to highlight of course, we have discussed earlier why sync thread should be not there, loop should be not there, we wanted to push in the fact that okay, this check is also unnecessary, fine. Now, if you want to further motivate it, you can also look at the situation that would happen with that let us I mean, let us proceed with the part you can think over this part also, right.

(Refer Slide Time: 06:07)

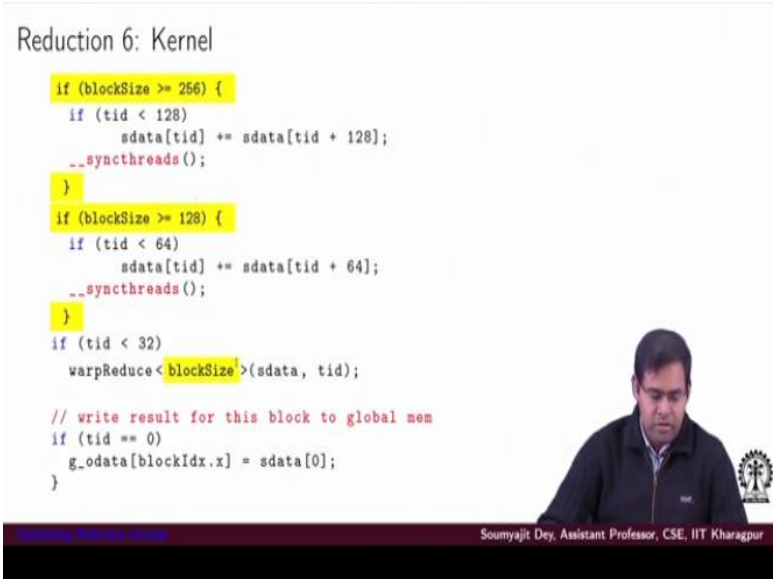


(Refer Slide Time: 06:13)



So, next the thing that we attempted was complete unrolling and there we brought in the concept of templates and with templates, our idea was that okay, block sizes are going to be multiples of 2 and anyway in the GPU we are limiting our block sizes, to the sizes here in the code that is up to maximum we are allowing is 512, okay. So, anyway I will not need the for loop since, I know that instead of the loop, if I come to a complete unrolling, I can write a sequence of if else blocks.

(Refer Slide Time: 06:41)

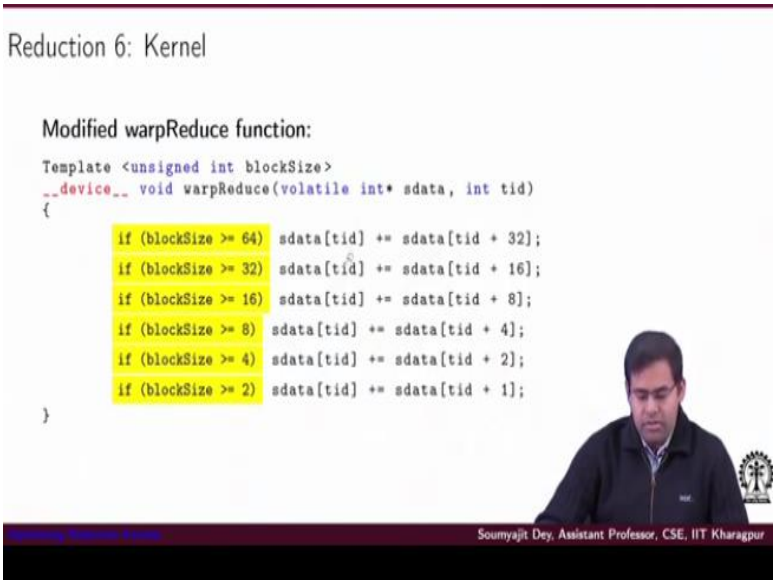


That if block size greater than 512, this will happen, followed by greater than 256 this would happen so, this would be the sequence here, right. So, anyway if the block size is 512 for that, I would get in here and execute this entire sequence of code, if it is 256 then, I will not need the previous if condition but I should start from here, so on so forth right.

So, the question that is also interesting here is; since the block parameter is templatized so, it is interesting to figure out that how I can make the system know that what is the block size that is going to be finally used and accordingly, the compiler can resolve these branches, this is something we did not discuss, so let us focus on that aspect here. So, just think if we are going to run the corresponding host code for this reduction, so here we have unrolled the complete loop, it is beneficial here.

Let me again point out because the maximum block size we are considering is limited to these values, right.

(Refer Slide Time: 07:49)

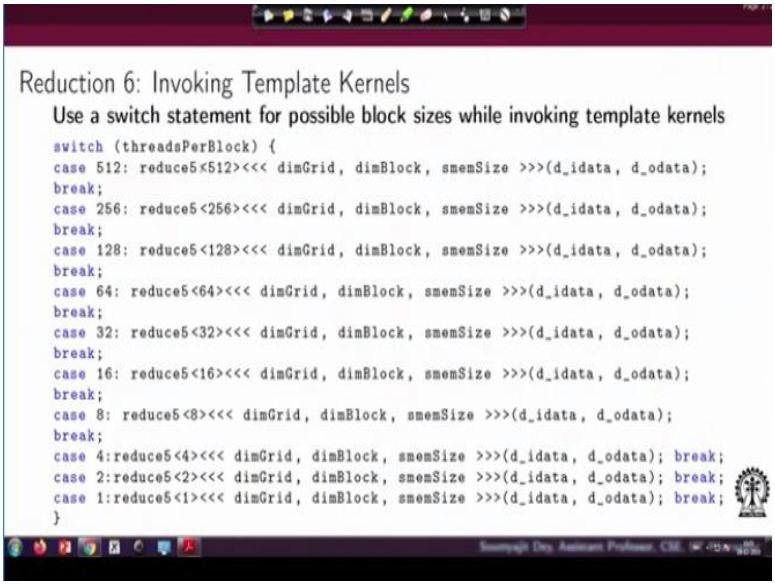


Now, this would be the; this would be also the corresponding warp reduced function, I mean if you just see that well again, when the tid is less than 32 just like previous case, I am calling a warp reduce function now, the warp reduce function is also being templatized with the block size. So, inside the warp reduce function I have this sequence of s data computations but now, I have an extra check here whether the block size is greater or equal 64, greater equals 32, 16, 8, 4, 2 like that, right.

I mean it is again coming because of course, it may so happen that in my block size is smaller than 64 in that case, I do not really need to execute this instruction right so, I will start from the next, so that this is the reason for which I will now push in this block size parameter here provided I am using the warp reduce function in conjunction with a templatized version of reduce 6.

I hope this is clear, I am just trying to push in the point here that if we are going to use reduce since in a templatized version with block size templatized, then correspondingly I also need to make a template function called for warp produced with block size being a template parameter.

(Refer Slide Time: 09:14)



The good thing here is look at the corresponding call from the host side for this kernel, so just a small correction here, this is the reduce 6 kernel, so consider this as reduce 6 yeah, so this is the host side call right. Now, as you can see here when you make a function call to this templatized function, this is your normal function call of the kernel, where for the function call of the kernel reduce 5 kernel, you are passing the dimension of the grid, the dimension of the blog.

And also the shared memory size that is; so that is a new function parameter which we are introducing here just to let you know that when you launch a CUDA kernel as a launch parameter, you can pass the block and the grid dimensions and along with that, you can also pass the maximum shared memory value, this kernel would be using, right. So, that is additional parameter that we have introduced here.

Now, observe that since it is call to a templatized function, you also pass; you can also pass the template parameter okay, I want to call it with a block size of this, so from a user or somebody from some other file, if I read what is the threads per block, then I can match the suitable case here whether it is 512 or 256 or 128 like that and accordingly, pass this as a template parameter.

Now, what is the good thing about this passing the template parameter; well, this is then resolved here that what will be the call. Now, when the compiler will be compiling this code, it will be ready with all these different versions of the reduce 6 kernel okay, so it will actually

be ready with these versions of the reduce 6 kernel because it knows that the block size values can be so on so forth.

And accordingly, using the block size value, it would have resolved these if else statements of the block size. Now that would mean, when you have the application binary using this template, you have actually got reduce 6 kernels different versions, where this block size greater than equal to some value, this if conditions has been resolved in the compile time. I hope this is clear; these are standard property of templatizing a function call or a class in C++.

You resolve dependencies, you met analysis and you are able to resolve as many possibilities as possible and accordingly, you create different binaries. So, when you are really executing the kernel, you are not going to really execute any branch instruction here that is the most important point, right. So, depending on the template parameter that has been passed, you will be executing a version of the kernel where it is already known from which if else block you start.

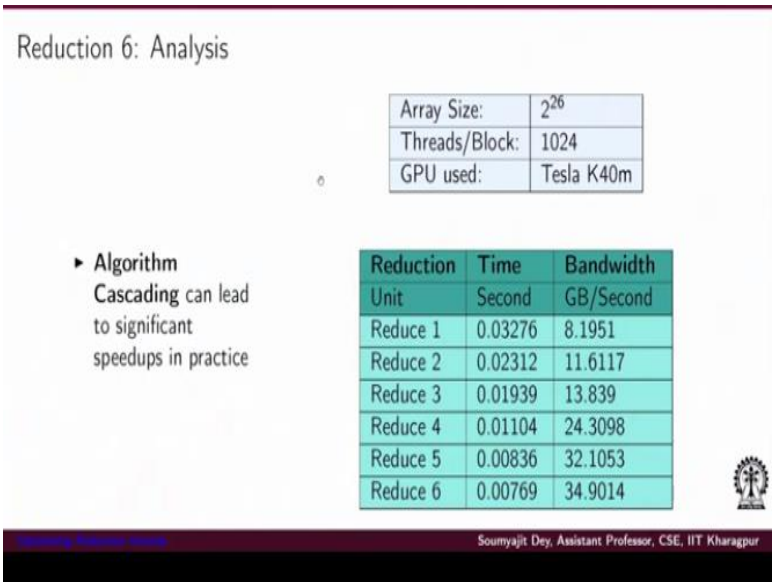
And you just do the corresponding addition computations, the parallel reductions computations for this kernel as well as for the warp reduce function, fine. So, that is how templatizing and complete unrolling achieves its full parallelism here. First of all, you have removed the loop and also you have removed the if else blocks during the compile time analysis, right.

So, this is also the advantage here just so, in the first case when we unrolled the warp 0 with the tid is 0 to 31, the good thing was; we are able to remove the for loop, we were also being able to remove the if condition because as we figured out, it is completely unnecessary, you can; because I am finally only interested in the warp containing that tid is 0 to 31 and that is the only warp which is being made to progress through this if condition, right.

So, when that warp is executing, I do not need to really do any final check and unnecessary waste time, let the warp execute without any kind of divergence because finally, I will only be using the data, s data tid is data 0, right that was I mean summarizing here for completeness, so that is how reduction 5 is done and this is how reduction 6 helps 0.1, you completely unroll the loop, 0.2 you resolve the if condition statically.

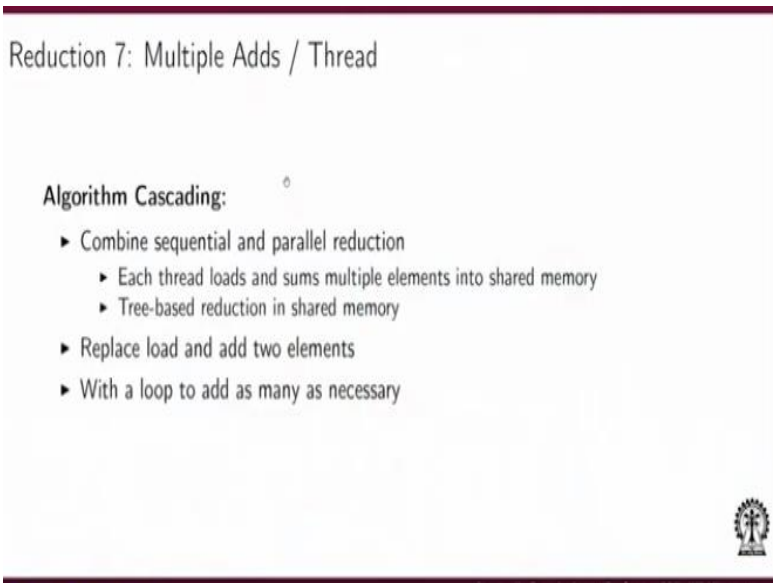
So, you really do not get the code diverging at these points both for the reduce 6 kernel and as well as for the call to the warp reduce function okay. So, this is very clear I hope now, with this incorporation with the switch statement based function calls from this the possibilities are already known, right.

(Refer Slide Time: 14:09)



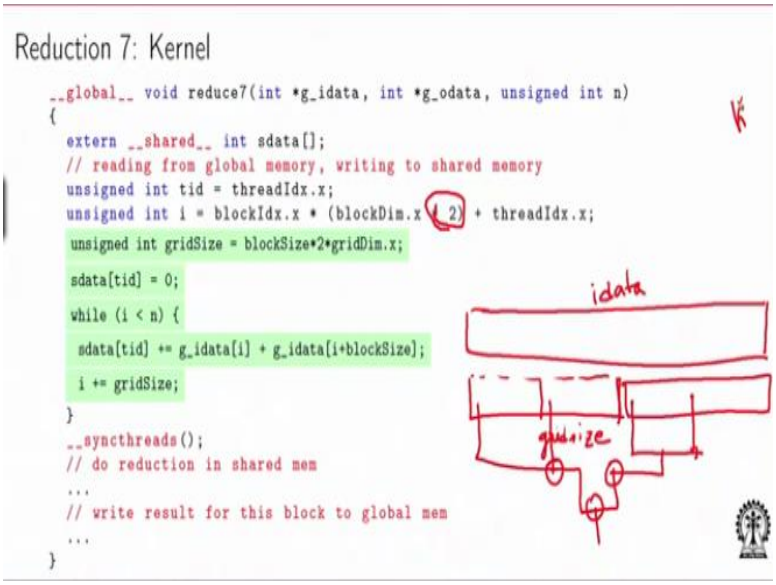
So, the last thing that we were discussing was algorithmic cascading, so if you remember the basic idea was that okay, we are going to increase the number of part thread activities.

(Refer Slide Time: 14:22)



Earlier in one of the optimizations, we increase the part thread activity by doing the fast addition after global load.

(Refer Slide Time: 14:27)



Here we are generalizing it that instead of doing the 1 addition, we will do multiple additions here right. Now, if you look into the code here so, let us draw a figure to better understand what is really is happening. So, consider that this is your actual arrangement of the data and if I do a mapping of the actual threads you are going to launch let us say, you are going to launch this grid of threads, okay.

So, since this is a 1d situation, so let us say this is your set of the data that you want to reduce but you are going to launch less number of threads, so your threads; the grid of the threads is up to this and let us say 2 of them are able to cover up for the entire situation, right. Now, inside the thread so, this is your grid size, this entire thing is your grid size, right and now inside your grid, let there be 2 blocks like that.

So, what we are doing here in the reduction step is as follows; so first thing you do is you compute the thread Id here and then you compute fellow i which is essentially block Id times block dimension times 2 plus thread Id, the 2 factor comes because of course, we are going to each thread would be loading 2 values at a time and then doing the addition, right. So, when I compute the i; the i is basically indexing how many addition operations I am going to do here, right.

So, there is the; is basically iterating over that space so, every time I am going to load 2 of the elements from the same block and do the addition, so when I compute this index i, I do a block Idx times block dimension times 2, so that is why this 2 factor would come in here and

then you multiply by and then you add the thread Idx for the offset, right. Now, this is your grid size, so grid size is again the grid dimension times block size times 2.

The factor of 2 is again here to take care of the fact that we just discussed that every thread would load 2 and then do the addition. Now, when the addition comes, so what essentially you do is; so you are trying to make each thread, add, perform multiple additions instead of one addition, each addition will take 2 data points, so you have this multiplied by 2 factor here as well as here, right.

So, when you add, you add across blocks right, so the first addition would be of 2 blocks considering this i as zero here right, so that is the addition then you do and then you hop to the next part of the data, so this is my grid size, right and I am saying that the total amount of data that I have is a multiple of the grid size so, you go to the next part of the data of the chunk that is of size grid size.

And then again, you load the data from the value at here as i and then shift by block size, right and this is how you progress, right. If there are more than, you can again hop to the next part for another grid size part and then again, there you do 1 addition and like that, so technically speaking, in each side, in each grid size chunk of data, what you are doing is each tid is computing an index i.

And it is shifting by a block and so it is essentially adding from i index plus 1 shifted with a block size, right. So, in that way if I am technically saying that okay, I consider a data with multiple chunks of grid size present, then for each tid I am effectively doing and let us say the number of such chunks is some k, then this i variable and n is of course, the total n right, so I am going to increment i in step size of grid size.

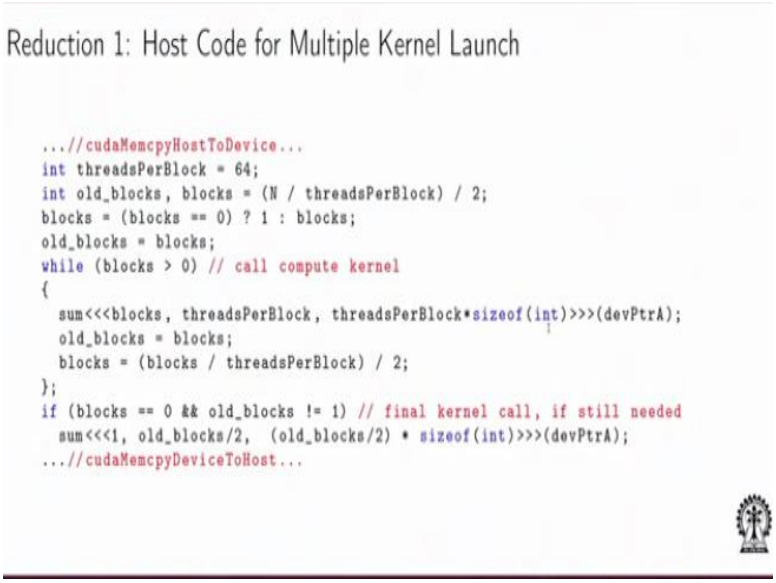
And since, the total data size is k times the grid size so, i would increase in steps of k and I will be able to do k number of additions here right, so in that way this is the general addition that if I have a large data block with this step, I am reducing the data block by doing additions on global memory data, then storing into the shared memory. So, with this step I am reducing the data block by a factor of k to a single grid size.

And then, again I am so essentially, what I am doing is for different grips, I am bringing the data and doing the addition across blocks, so it is not only the case that I am bringing data from different grid sizes also, I am doing 1 reduction by taking data from 1 grid and adding with a block size offset. So, I hope this is clear, in the original version you are considering only this as your width and you are doing 2 additions, right, you are doing 1 addition for 2 data points at a block width.

But now, what you do is; you consider the actual data of much more size by a factor of k, so here you take a; you not only do an addition inside the grid size but you do such k number of additions across different chunks of grid size and then you reduce this entire thing to a smaller block of this size, the grid size and then go for the shared memory based reduction, right.

So, this is how the thing progresses here also, there is something important which we have not discussed till date that what is the way the host code looks right, because we have discussed that okay, I can have multiple blocks of data for each block how the kernel will do the reduction, right but in general we have always said that okay if there is a lot of blocks, then the host code will take care of launching the kernel reducing the data in sizes of blocks and then keeping; I mean keeping and it will just keep doing this on and on until you get to the final block and reduce it to a single value.

(Refer Slide Time: 21:55)



So, going back and revisiting our host code, so this is the code that actually takes care of launching multiple kernels across different blocks. So, consider that we have threads per

block equal to 64 and we have 2 of these variables; old blocks I mean, the blocks previously to the reduction step and the blocks right now, so what we do is the number of blocks is total data size n divided by threads per block divided by 2.

Because of that global addition, we are considering the global addition scenario; 1 addition per thread right, so first thing is we initialize the blocks value and put that value into the old blocks value and then inside this while loop, what we are doing is; we call this kernel sum with parameters blocks threads per block and the amount of shared memory that we want to pass it.

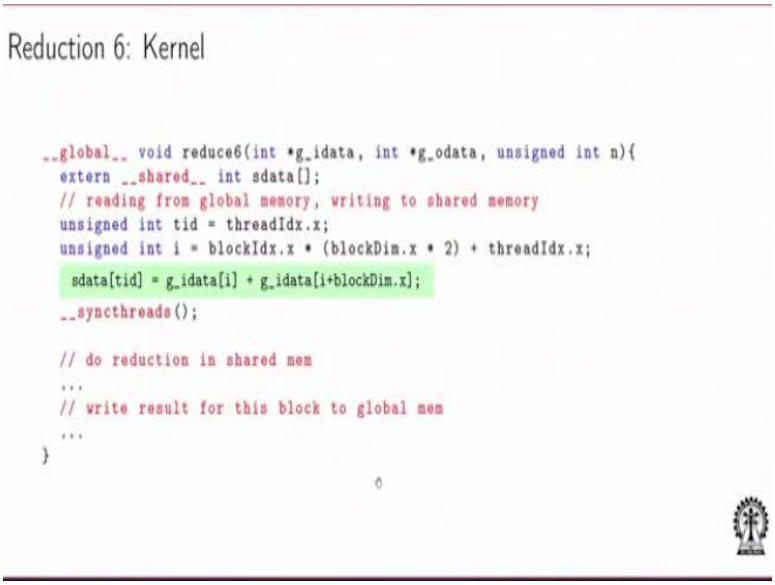
So, this would make a reduction and then you have this number of blocks which you store as old blocks and definitely, you have to reduce the number of blocks here now of course, we know the reduction factor is you reduce the number of blocks by the threads per block because every block gets reduced to a single value and because of one global memory addition get further reduces by a factor of 2.

So, that will be the new value of the number of blocks after one step of reduction by this kernel. I hope this is clear so, you are calling this kernel over blocks and threads per block parameter and finally, after the kernel returns, you have to recalculate the number of blocks that are there and that would be since each block gets reduced to 1 value so, blocks divided by threads per block.

And if you consider the situation that each thread before going to a reduction step is doing 1 global addition, so that would blocks actually also reduced by a factor of 2, so this is the final value of the updated number of blocks, you are going to keep on computing this as long as the number of blocks is greater than 0, if its final equal to 0 or this whole blocks is not equal to 1, then you launch the final kernel to reduce the remaining block to a single data point right.

So, this is the host code that we skipped earlier so, with this now we have finished our discussions on the different possible reductions.

(Refer Slide Time: 24:24)



And so the final reduction, I will just reiterate here what we did was that we actually increase the amount of work to be done on the global memory by considering that okay, the grid size we actually hop over the grid size chunk, get into the other chunks of data and also perform an addition step block wise with a stride of block size for each thread, right each thread is computing; is doing for its value of; unique value of i that the thread would compute for a corresponding block right.

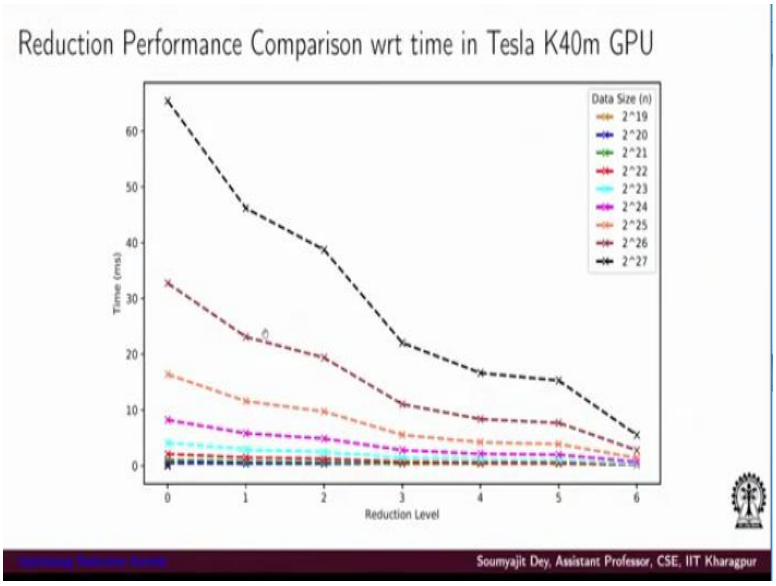
It will consider that corresponding location in the global memory and the next block and do the addition right, so I mean I hope this is clear, the part thread activity which is very important. So, the first point as we mentioned was that each thread would be hopping over grid size chunk of data and doing 1 addition but the also, the important thing is what would it be doing inside 1 grid sized chunk of data, what it would be doing is; it will be computing 1 offset parameter i.

So, essentially it knows in which block it is, it will multiply by 2 for the global add considering and then multiply by block dimension get into its own block, then it will be get into this thread Id for that position and the corresponding offset position in the next block, it will be doing the addition inside a grid size chunk of data and then would again shift to the next grid size chunk of data.

This is how it would carry on the computation here okay, so with this we will add; we will actually end our discussion on performing addition in parallel to be specific and as you can

see that this addition can be other reduction operation, the same methods would actually be holding up here.

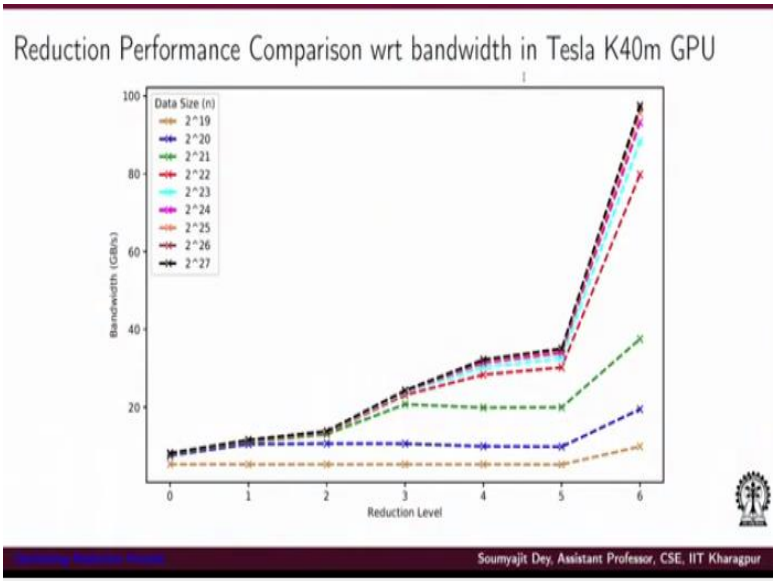
(Refer Slide Time: 26:23)



So, thank you for your attention, in the next lecture we will be actually go into a different topic some other algorithms, now just for your reference we provide some data here that how parallel reduction performs for the varying data sizes, so as you can see we are varying the data size here so, this black line has got the highest data size, so and for that we are showing that with the application of each reduction step, what is the decrease in execution time that we get, right.

So, this one gives is the actually I am highlighting the one with 2 to the power 7 data size because you can see that maximum speed of you are possibly getting here and the experiment is done on Tesla K40 GPU.

(Refer Slide Time: 27:10)



And similarly, now if we calculate the bandwidth utilization which is a primary reason for getting the speed up, as you can see for this 2 to the power 27 size, we have the maximum bandwidth utilization of course, it is closely followed by the other data sizes here okay, with this we will end our discussion today, thank you.