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

Module No # 03
Lecture No # 15
Multi-dimensional mapping of dataspace; Synchronization (Contd.)

(Refer Slide Time: 00:27)

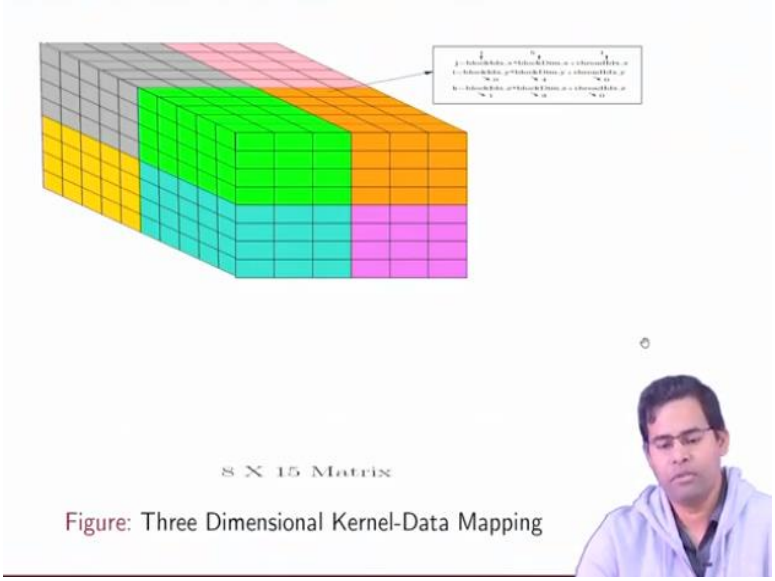


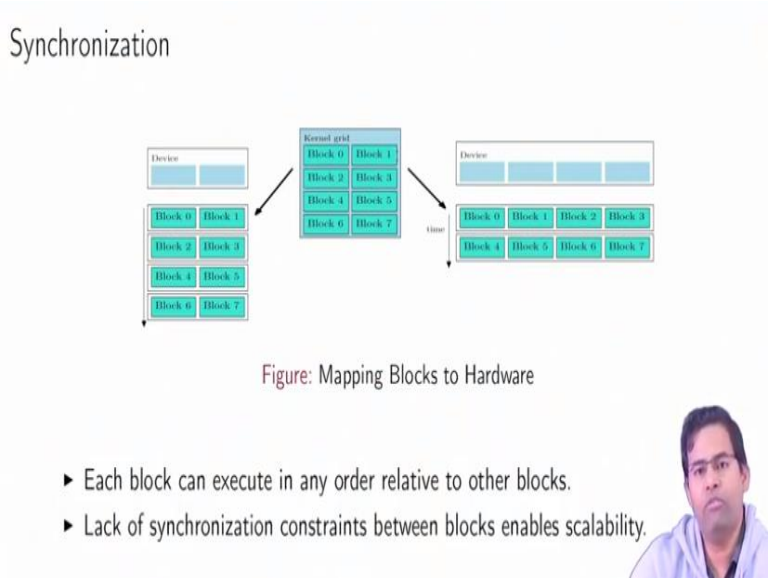
Figure: Three Dimensional Kernel-Data Mapping

Hi so in the last lecture we have been discussing about multidimensional mapping of data spaces like how to choose may be suitable ways to map high dimensional data in terms of the kernels like how to choose a suitable access expression for the different data segments so those were just come initial thoughts we discussed and we saw that what is advantageous in each case and of course this would be more clear in future.

When we discuss more examples and provide more assignments with respect to different kind of handling different kind of high dimensional data spaces and creating what we call as access expressions. That means how to really how CUDA's are thread really identify what is the minimum elements it is going to warp one and just to recap that this was based on a threads knowledge of its block id thread id variables it is being able to use those variables to create suitable access expression for different data point it was going to work on.

And as we saw that it is very much of function of how do you define the grid and the block that means whether you define a 2D block or 3D block and a 2D grid or 3D grid and accordingly the access expressions are subject to change. So this is something you can try with more examples and we will also try to show that with examples in terms of assignments later on.

(Refer Slide Time: 01:55)



Now we will more on to another important topic of synchronization among threads now first of all what is synchronization? In general with respect to parallel programming semantics synchronization are essentially means some points in the program where we definitely know that okay this is the point where the threads have computed upto this point and once this point of computation is reached this is where every thread should reach and together or if some thread reaches this point faster and then some other thread which is this point and the thread which is this point faster should wait for the other threads to come up to this point and then again they can go forward.

So is basically a primitive through it is okay so that is why we call it as synchronization point because that is where every thread should stop maybe they can collaborate together they can ensure some sanctity of reads and writes on the data variables and then they can move forward with further computation. So we will see that I mean of course again this is a high level idea which can only be clarified through some major examples that we will touch upon.

So let us try and understand how synchronization can really be enforced on different threads which are computing in a CUDA program. So as we know that the kernel launches a grid of threads the grid contains a set of blocks and this blocks contain threads back to internally and these blocks contain threads backs to internally and these are actually getting schedule inside the SM's right.

Now each block can execute in any order relative to other blocks these are very important thing so if you take an example that suppose you have a important device and you have got 2 devices in your system 2SM's right and may be in SM 1 you have got the blocks 0, 2, 4, 6 schedule in SM2 you have blocks 1, 3, 5, 7 schedule. So overall your kernel has launched this many 0 to 8 block ID blocks right.

If you have the 2 devices there can one there can be possible way in which these devices are processing the blocks of course this is not a unique way there can be other ways. Like for example suppose I have 4 devices I can have a mapping where block 0 and block 4 are map to device 0 device means the SM's and like that. So there can be various possible mappings of blocks in the SM's each block can execute in any order relative to other blocks.

That means once I have launched all the threads in a grid there is no relative specific execution order like which thread will execute faster with respect to some other. I do not have any control over that unless I put in some extra primitive. So every block of thread can execute at its own phase that is the point that we are trying to decide here. Essentially the CUDA's the GPU as got a complex scheduling hardware there is a 2 level scheduling hardware and it depends very much on how the hardware's is going to schedule blocks there is some high level knowledge available in open domain about that.

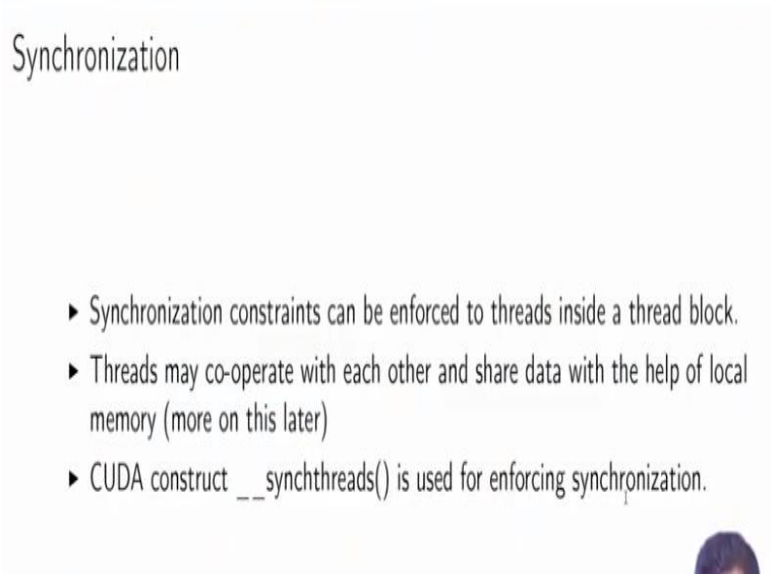
However every detail is available till now in open domain is very much part of the implementation of the GP but what is known in open domain is that there is a 2 level scheduler. So at some levels is decided which blocks are going to be part of which SM and inside the SM's we have scheduler to decide how to execute the blocks. We will get into more details later on but for that time being we let us understand that as a the programmer you do not have any control over which blocks executes in which order.

You have launch the kernel that would initiate all those set of blocks that will initiate you this entire grid then it is part of the hardware's architectures hardware schedule job to execute them in any order. However certain notions certain scheduling notions gets enforced which we will discuss. So the important take away from this slide is that the block can execute in any order creative to other block.

And there is lack of synchronization constraint between blocks there is a good thing because it enables scalability. So as a programmer you cannot enforce synchronization across blocks of code that means I cannot enforce the block 0 will execute only let us say after block 5 execute or similarly blocks 6 executes only after block 1 executes I cannot enforce that this actually is considered a good thing with respect to the hardware because since this cannot be enforced unless you do some fancy programing.

Again I would repeat if you write as normal vanilla CUDA program you cannot enforce such a thing and so the hardware is at full freedom to execute the blocks depend on the amount of SM or SP that are available that are free and it can schedule them with full freedom and actually provide the maximum through put that is possible that is the good thing about not synchronizing across blocks.

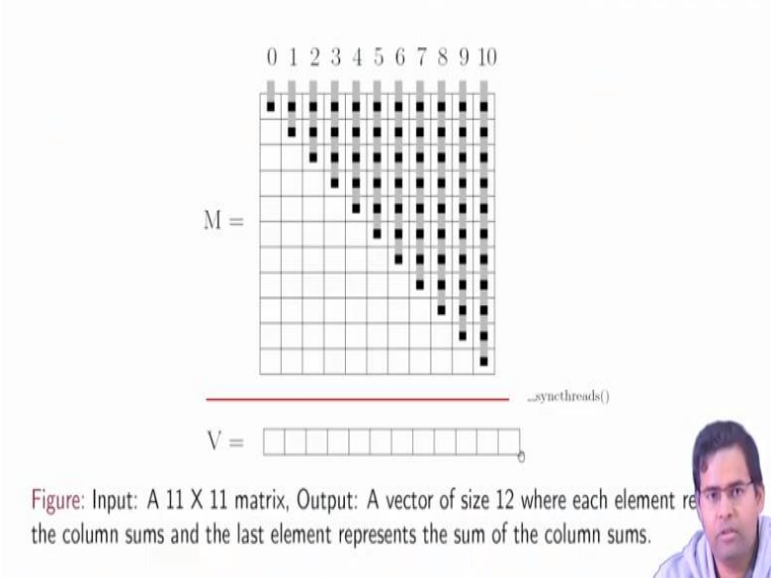
(Refer Slide Time: 07:13)



However since that is the question of course understood that synchronization can be done among threads blocks inside a thread blocks that is the very point we are trying to roll in here. So when you launch so I will just repeat this is very important when i am launching the kernel all this blocks are getting thread blocks are getting launched they will be mapped to the individual SM's and inside the SM's the order in which the blocks execute or across SM's the order in which blocks execute is beyond the programmer control.

So I cannot really enforce us synchronization constant among the blocks however as the programmer what I can do is I can enforce synchronization constants on threads inside the thread blocks. So inside a thread block the thread block the threads may cooperate with each other and shared data with the help of local memory more on this we will see later on. First of all let us understand how synchronization among threads can be enforced. So for that you have the CUDA constants sync threads which is used for enforcing synchronization okay let us see some examples on that.

(Refer Slide Time: 08:21)



For example let us take this sample computation here your input is an 11 cross 11 matrix and you want to output a vector of size 12 where each elements represents the columns sums and the last element represent the sum of all the column sums. So you want the threads here to compute I mean it should not compute anything it just reports the value for column 0 here thread with ID 1 would do a partial sum of thread of this position and the next position.

Basically the position basically the entries in M I would call them M01 and M00 and M01 and this is M11 the sum of these 2 locations that I should compute and put in here thread with id 2 should compute this sum and put in here and like that. And finally somebody should be actually computing the sum of this entire array and putting it to this location right. So this is what we want to do.

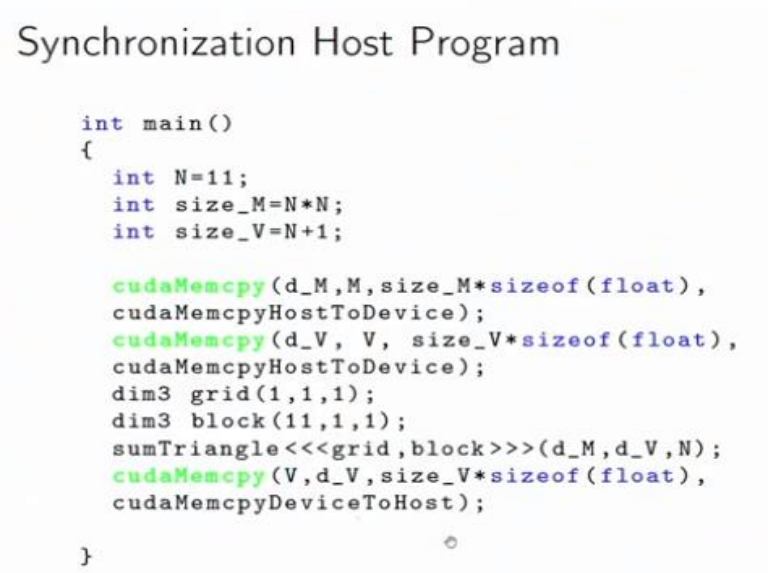
So now we need to understand that why synchronization would be necessary there synchronization would be necessary because we want all the threads to execute the same kernel code. However the amount of partial sum of computation that each thread is going to do is different right. Because if I pick up this thread the one I am marking here if I pick up this thread I want it to sum only up to this point right.

So as you can see that although this threads are launched with different id's the amount of work the kernel to make them do is different that would also mean each of the threads are going to finish their execution at difference points of time right. Now if this threads are going to be inside a single thread block which is the case here then I need to make the thread which computes it is job earlier I need to make it wait for the other threads which are going to complete their jobs later on right.

Why is that so because unless all this intermediate values are ready I cannot ask some thread to do a summation of all the entries here right because I want this thread to compute the partial sum of these points are put in here. I want this thread to compute the sum only upto this point and then put it here like that and only when this is done then i can go and do the computation of summation of all this values and put it here right.

Which means I need to split this computation across different phases in phase 1 all this partial sum computation should be done and this results should be updated only after that I should like to have a sink so that I have to enforce for that I would like to put this sync threads here only after the phase 1 is completed for all the participating threads should some thread go on and work with these values. So if I do not do a sync thread here something random can happen we will see that.

(Refer Slide Time: 11:40)



So for example let us look at the code so this is the host side program for synchronization so you have 2 CUDA main copy commands using which so essentially you are transferring host to the device the 2D array M and the array AV which is your going to be a result and from the dim3 primitives that you can see for grid and block you are just launching a single block and that block is also you are launching a single block because as you see these entries are 1 in the grid declaration.

And in the block declaration you can see that you are just launching these 11 threads right so it is 1 dimensional block and that is the only block that you are launching and with this launch parameters you are launching the kernel sum triangle and this kernel is going to work 1. Now the device side array dm and dv right so N is the size of the input matrix each dimension right and once this kernel execution is done you want to copy back this content of dv to the host side 1D array V. So this is your simple host code here now let us look at the kernel code.

(Refer Slide Time: 13:00)

Kernel

```
__global__
void sumTriangle(float* M, float* V, int N){

int j=threadIdx.x;
float sum=0.0;
for (int i=0;i<j;i++)
        sum+=M[i*N+j];

V[j]=sum;
__syncthreads();
```

So kernel code is going to tell you what is a part thread activity and let us have a look what is part thread activity. So the first thing you will do is you will consider what is the thread id. So in this case since I have just a single block and that block is also a 1D block. So I just have to care about the thread id x dot x variable and this variable is going to give me the this variable is practically going to give me the column on which the column of 2D array on which I am going to operate right.

So let us pick a sample value of J let us say I am going to talk about the column with id 5 so that means I am talking about the thread with id 5 starting from 0 right. So now all that is going to happen is if you look at this picture I have in like so I am talking about this thread so it is going to sum up 0, 1, 2, 3, 4, 5 these 6 values starting from 0. So you compute the sum of the quantities here which you access through this access pressure M.

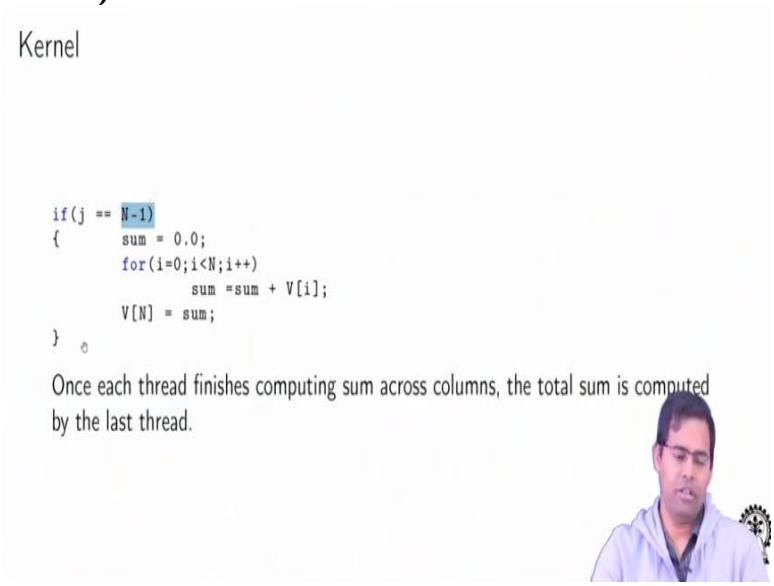
I is the row number from 0 you go upto I less equals j so you go upto j – 1 right and so you have so you complete that many rows and then come to the jth column so in that way with this expression if you look at this expression I times N + j for different values of j you are able to access these different locations right. Because fundamentally this is all sequentially located right so with that kind of an access expression you are able to sum up all those elements in the column with id j but that is also up to first j elements right that is what you are summing from the zeroth through j – 1 that is the j elements right.

So once this is done you store this sum this thread will store this sum in the location Vj of the output array. So that means once this computation is done it will store at this location of the output array. Fine and after that we have a sync thread like we are discussing that ok once every thread has computed this partial sums and they have stored this results in the output array there is a sync thread so that means every thread stops here why?

Because I want to compute the summation across this array and stored that some at this point only when the job of all this different threads are completed right. That is why I will put that sync thread after the partial sum computation and storage in V and before that is summation starts according across V because I want to ensure that before I do the summation across the entities of V all the entries of V are ready right.

So that is why after computing this Vj in 3 I put the sync thread once all this entries are ready then I get into to ask that one of the threads that okay now you compute the sum across the kernel. So once each thread finishes computing the sum across columns the total sum is computed by the last thread. So essentially how do I choose which thread is going to do this job as you can see that we have launched this many thread that 11 threads we have launched and the last thread.

(Refer Slide Time: 16:47)



So as you can see we so this is the part thread activity for this kernel and only the thread with ID in -1 would enter into this block right only the thread with if n – 1 will enter into this block and then what this thread is going to do is it is summing up the entries of V and storing it into the

location Vn. As you can remember that V has got one more space extra space so V starts from V0 and it is upto VN.

So only the thread whose ID is j would be equal to n – 1 so that is essentially the thread which is computing this entry that will get into that if block the other threads are not get into the if block but only that thread would do the summation and then store it up here. Now the question is then what was the requirement of sync thread. The requirement was there because we want the following to be ensured that when this summation is being done on the output array.

All the VI entries in the output array should be ready now which may happen that you run the code without the synch thread but you still get the result. Question is there is no guarantee that you will really get the result you may get the result may not get the result why because without the sync thread then we are assuming that before the last thread starts doing this computation everybody else has finished it is not guaranteed.

Now the question is when do I really not have the guarantee okay if we are working with small number of threads where the threads are all part of the wall then I can have this guarantee but as we have discussed that in a general setting when I have very large number of threads in a block I have got very large number of threads in a block then I cannot guarantee only when the threads will be part of a single warps then I know that the threads will execute in lock step and then I have a guarantee that okay when the last thread is going to finish by that time the previous threads which are in the same wrap same warps as actually finished.

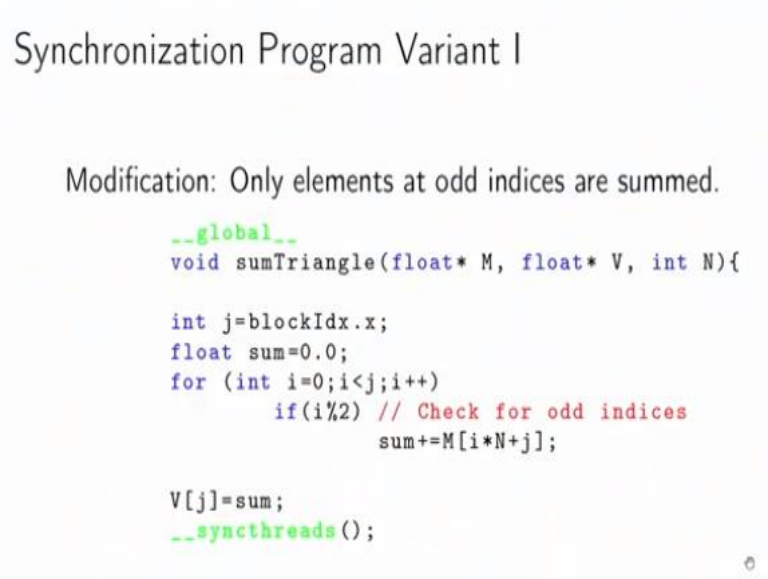
Bu tin general that may not be true only in the case when I am having multiple warps which are created inside the block right. So just to highlight your example if you are running this code with small number of threads like this the threads will be part of a warp and you may not need this sync thread. But if you are running this code on a big thread block which is going to launch multiple warps then you do not really have a guarantee because then what is going to happen is each of the warps are going to schedule threads in different ways will come to formulizing that later on.

But just to understand that I am trying to remove a notion here that you may run this code on a small setting of a block and see that well yes without the sync thread still things are fine. I am

just trying to say that it may or may not be fine it depends on the size of the block and we will see that exactly in which case it is good and which case it is bad. But in general you need to appreciate that when I am running this threads I do not have any guarantee that this thread is going to finish last or this thread is going to finish first it depends on how many threads are lunching.

If I launch this small number of threads we are talking this small number of threads just for a figure representative purpose it may happen that i do not need the synch thread but for a significantly large number of threads I do not have any guarantee. So anyway from a programmers point of view I need to ensure that all this entries in the v are really ready that means all the threads have really done their job before this last thread goes to compute this entry right.

(Refer Slide Time: 21:18)



So for that purpose I will need to put this sync thread here fine so that would about this small program and then let us look at some other variants. So suppose I hope just to recall I hope the idea is clear that I just wanted to put this point across that it may not be the case if we are using a very small number of threads or we are launching a small number of block but in general for a significantly large block and in general unless I put this thin thread here I am just trying to summarize here.

And I do not put this thin thread here then there is no guarantee on how fast each of the threads should compute. So to enforce the constant from the programmers point of view I would like to have the sync thread here fine. So looking at other variance of the program so now let us consider the situation that we only sum up that elements that the odd in this s.

(Refer Slide Time: 22:19)

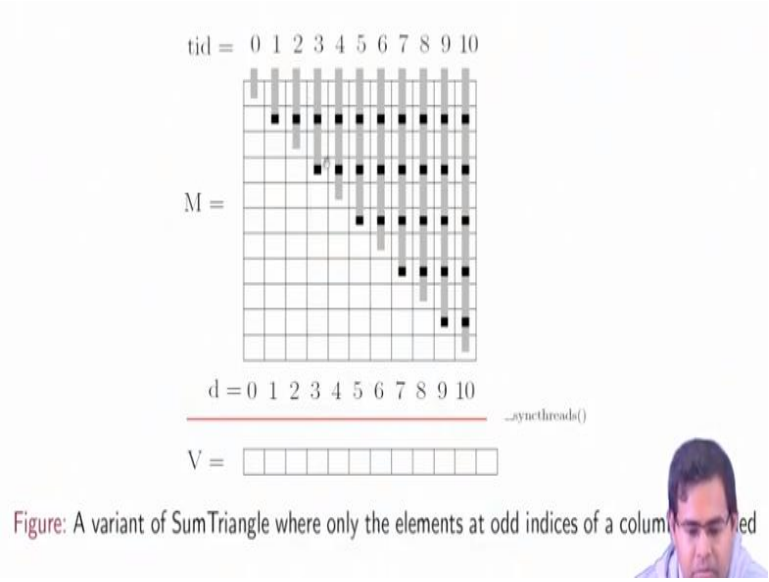


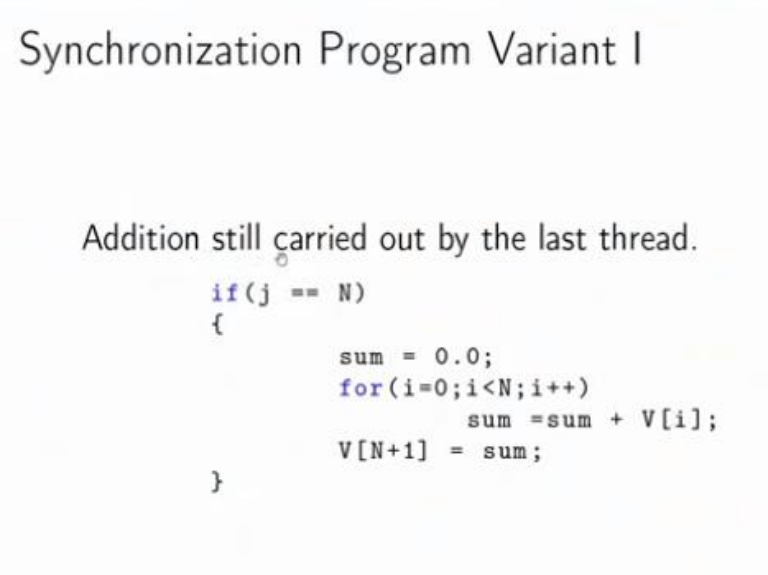
Figure: A variant of SumTriangle where only the elements at odd indices of a colum ed

So that would mean if I look at the example so I am just summing up the elements at this indices here. So at index zeroth element the third row element have not including the elements are the row number 0 or row number 2 like that right. So again including the first 2 element third first row and third row included here first row and third row is included here first, third and fifth row is included here. Again first, third, fifth and seventh row is included here so we are going in a one hop basis.

If I am trying to do such a variant of the same computation how would really the kernel look like. Well it is simply a matter of putting in a choice statement here so essentially I have the same code so I believe this should be thread id here just a minute sorry for this yes essentially is the same code okay. Let us just review the earlier code once again so this was your original code and if you come here is basically the same but you are doing the sum only for the odd indices.

So you push in a small check if I percentile 2 so if that is 1 that means you get in for the odd index rose and otherwise you do not get in and only in those cases is the sum and in that way everything remains the same right fine.

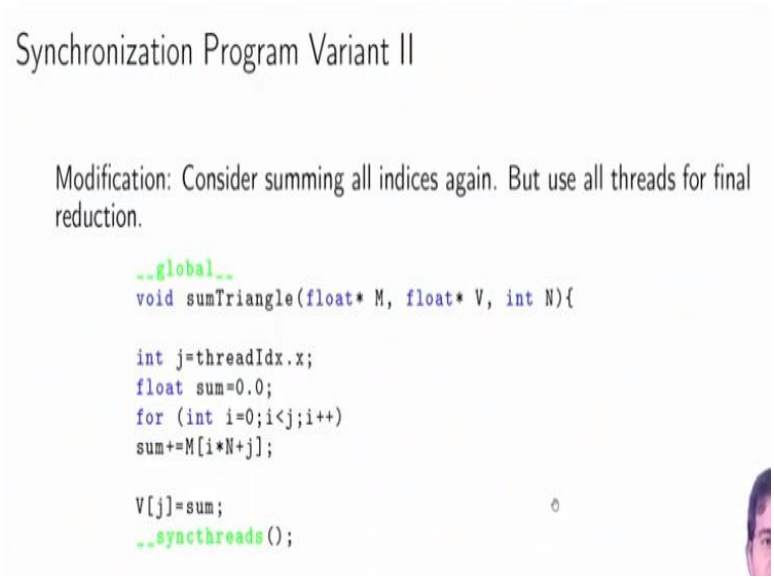
(Refer Slid Time: 24:19)



So rest of the code is same your addition is still carried by the last thread so it is all the same here. So you have the same thing same setting but as we are explaining every thread is hopping over an element and again we need the synch thread here if I have a significant number of threads all of the threads are not pushed in into a warp. So you do not have any control over which thread is progressing at what speed so after their each of their partial sum computation again I would definitely need the synch thread here to ensure that yes every thread as done the computation properly I mean they are all finished.

And accordingly now we will instruct once every threads reaches this point this is the synchronization point only when they reach here they again restarted for a fresh run. And again the last thread starts summing up this entries to compute the last entry.

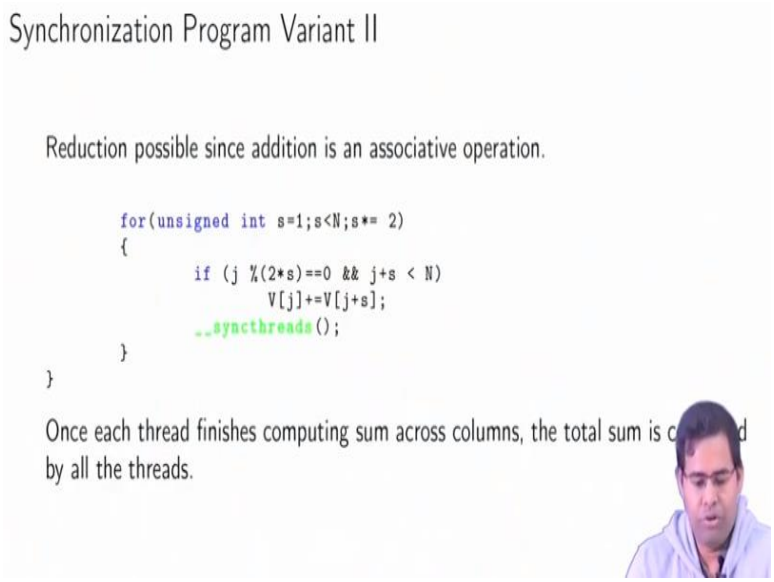
(Refer Slide Time: 25:24)



Now consider a different variant of the program again we are summing all indices like the first example like we are not again hopping over the rows we are going to sum all the rows. But now we are trying a modification that instead of using a last thread for the final reduction that means for doing the final summation computation we want to do the final summation computation in a collaborative way by using all the threads.

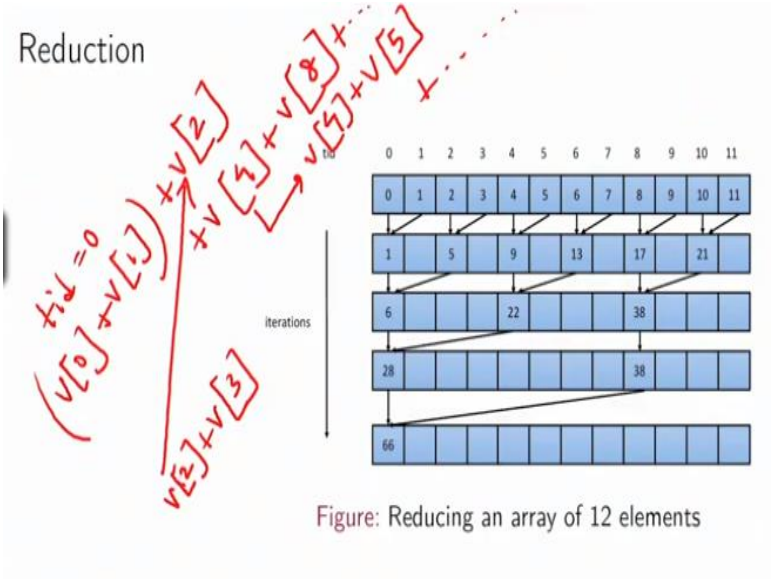
Why because as you can understand if I give that job to one thread for a significantly large block it may require lot of time but I may want to speed it up by distributing the warp among all the threads because finally why do I leave it for one thread there are many other active threads which are simply sitting there idle right.

(Refer Slide Time: 26:18)



So again the first part of the code is again exactly the same for this piece of code let us first try to understand what is the activity that each of the threads are doing and we can this activity can be actually visualized graphically in a better.

(Refer Slide Time: 26:36)



So if you look at this example here so we have the thread id's all plotted like this in the X axis and we are showing what each of the threads are showing in each iteration right. So let us just follow 1 or 2 iterations and things will be very clear. So first of all in the first iteration of this loop every thread is going to enter this loop with an s = 1 j is the thread id now as you can see in the if condition we have j + s is less than N.

So since j is the thread id and s is 1 so this will be satisfied by all but the last thread right so for all the id 0 to 10 this is fine right is 11 our example here but the first part of the condition is 1 so I am doing essentially a j percentile 2 = 0. So that is only going to satisfied by the threads with the even indices right. So all the threads in the even indices is going to do something in the first iteration of the loop and what are they going to do they are simply going to add the entries in the array at the location vj which vj + 1 right.

So essentially for thread id 0 is going to compute V0 + V1 and store it in V0 so that what we have V0 + V1 and store it in V0 by thread id 0. Thread id 1 is odd index does not do anything thread id 2 is even index does V2 + V3 and stores it V2 and similarly everywhere. So at the end of the first iteration of the loop half of the threads actually collaborated among each other and they have successfully computed the partial sums at tid of 1 is denotes the tid.

So with 1 hop half of the threads are collaboratively computed the partial sums so we can understand how things are going to proceed here. In the next iteration of the loop another half of the threads are going to be active and they are going to increase the tid by double that is s = 1 to s = 2 and they are going to do the partial sum computation of entries this and this right. So essentially if we go back to the code in the next iteration it gets multiplied by 2.

So we shift from tid 1 to tid of 2 and the thread id 0 will again satisfy the condition but it will get in the loop and compute a V0 = V0 + V2 here right. So if you look into the code initially we did V0 = so if I just look at what thread id 0 is so here it computed V0 + V1 this value got stored in V0 and this is how added up with the content of V2 and then next it will get added up with the content of in the next iteration with before in the next it will get added up with the content of V8 and in the process the V2 that is getting.

We already have V2 to be equal to V2 + V3 if we look at V4 I have already have V4 + V5 and then V4 + V5 + actually V6 + V7 so in that way I have the sum of to V8 and the other elements will also come in right. So in this way as you can see that the entire computation will make a progress and all the threads collaboratively compute the sum. So just to repeat in the first iteration half of threads are simply adding their own position with the next position.

So in that way I have half of the partial sums stored here in a next iteration I have further half of the threads that is one forth of the threads computing this locations along with the location at a stride of double space that is S = 2. In the next iteration we again have a further half of the threads which is actually summing of this location along with the stride at an even double space so S = 4 in that way I am able to compute the sum for the entire array.

So you can just write if you are trying to prove the correctness of the program if you can just write at the final location sum is equal to this 2 location sum this is again equal to sum of this 2 locations and this location is again equal to sum of these 2 locations and you can go back like this. And finally you get this expression where this locations is equal to the sum of all the individual locations right.

So that ensures the correctness of the reduction and we can actually prove that this simple optimization gives me the correct result but it actually engages all the half of the threads together in the computation. Unlike engaging 1 thread to do the entire computation. Now observe something very important here after this first iteration I have got all the threads to gives me the value of its own locations data plus the next location right.

Unless all this individual computations are finished none of the thread should be allowed to go to the next iteration because then there will be inconsistency. For example let us say thread 0 will progress to compute this plus this but may be thread 2 has not completed this plus this operation in the previous iteration right that can happen because of the situation. We discussed earlier that in the GPU I do not have a guarantee that threads are progressing together unless they are part of the same world right.

Unless they are part of the same warp threads are not progressing together even if they are part of the same warp based on the conditional execution that warps may diverge and there will be things that we have to discuss that for the timing we can just assume the treads executing inside a block are progressing at their own speed. Since we do not have control over there execution speed because of the hardware scheduling taking care of that I cannot really guarantee that 2 threads doing this partial sum computation are progressing at this same speed.

But we can see that we have the requirement here unless each of the threads are able to do their computation for some specific value of the stride none of the threads should allowed to go to the next level of computation for the next value of the stride I cannot let S progress and all the threads progress to the next iteration of the loop unless all the threads complete the previous iteration of the loop and the way to do that is very simple just put a sync thread inside this loop here.

So once every thread is guarantee to a execute one iteration of the loop only then all the threads together go for the next iteration of the loop. So I will just repeat once every thread is guaranteed to compute executing 1 iteration of a loop only then we should let all the threads together progress to the next iteration of the loop and this is how the things should keep on going and the computation would keep on progressing and finally we have all the threads computing together and giving me a final result.

So with this we would like to sum up our explanations on synchronization of course we will have some more examples in future which will involve synchronization but this is just to give you basic introduction of how to and where to put sync thread primitive in the code. So basically we have to understand at what are the point in your kernel where you need to guarantee that all the threads of thread block have actually done their computation before proceeding to the next line of computation.

If there are such algorithmic issues where all the threads are going to use results computed by each other at some point of computation or from starting from that point of computation there as to be sync threads statement for synchronizing the threads inside the stride block. So just to summarize threads inside a thread block can synchronize you have to use the sync thread primitive but the reason here to use it just to sure is that you do not any guarantee that what speed the blocks and the threads inside the blocks are progressing inside the GPU how is the scheduling done. We will investigate more into this from the next lecture thank you.