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

Module No # 03
Lecture No # 13
Multi-dimensional mapping of dataspace; Synchronization

Hi everybody so today we will be starting with the fourth topic in this lecture series and today we will be focusing on this idea of multidimensional mapping of the data space I mean for a CUDA program and also the related concepts of synchronization among parallel CUDA threads.

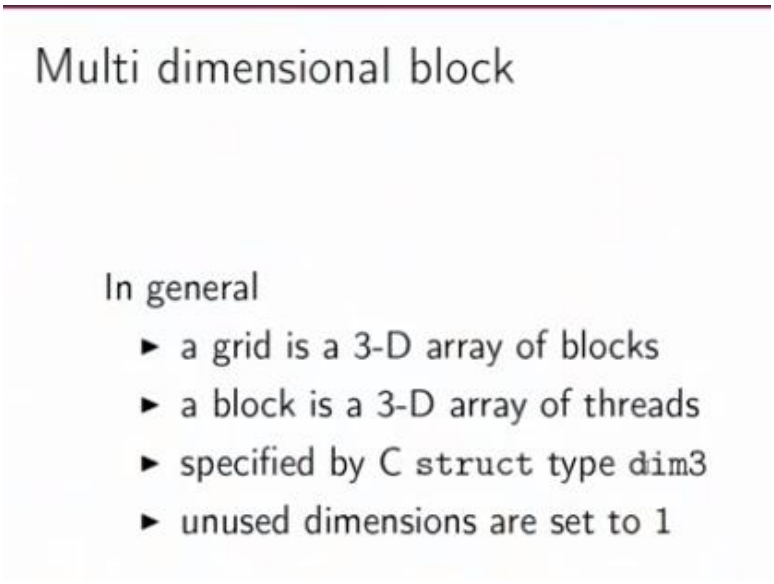
(Refer Slide Time: 00:47)

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

So just to get back into the overall prospective we have discussed this idea of basic computer architecture and then we have migrated to more detail ideas on GPU architectures we have introduced the basic syntax of CUDA programming from this point we are trying to get into some more finer details of CUDA programming like how to handle complex data structures how to synchronize threads which are operating in parallel and all that.

(Refer Slide Time: 01:19)



So that is what we will focus on here so if you remember our lectures on basic CUDA programs and examples of matrix multiplication and similar kind of programs that we discussed about. There was this notion of kernels getting launched and as the launch parameters we are supplying certain parameters which is the number of blocks and number of I mean number of threads inside the block.

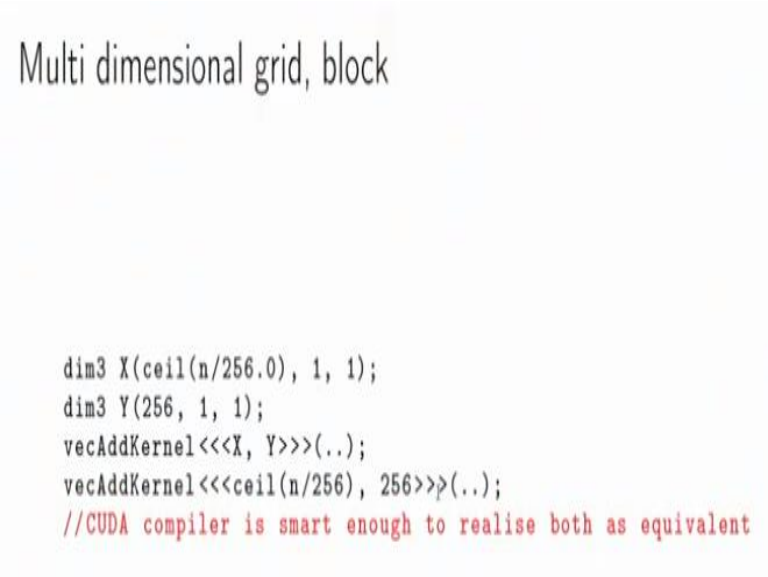
So that is how we define them and we also said that this is something we will take up as a future topic so this is where we will discuss it finally. So when we say that CUDA kernels is getting launched essentially I launch a set of threads now these threads are packed into a finer element which is known as block so essentially you say that I will launch multiple blocks of threads and this entire arrangements of threads is called as grid.

So in general when I am talking about the overall idea of CUDA threads that are getting launched I am launching a grid of threads the this arrangement of a grid is kind of the it can be defined like this that I can say that essentially a grid is nothing but a 3D array of blocks. So inside a grid I have entities known as thread blocks or blocks and I can pack a box in 3 dimensions we will see this 3 such examples and then in the similar concept will repeat inside blocks.

So I can say that a block is nothing but a 3D array of threads so overall I have the threads arranged as 3D arrays I mean in 3 dimensions and packing's of data structures called blocks and

the block can again be arranged in 3 dimensions into packing's which are finally defined as grids and these idea of 3 dimensional definition keys corresponded to by a specific data structure type which is a specific structure which we define in CUDA in the extension of in the CUDA semantics and we call it as a structures called dim3. So if I have not really interested in extending the dimensions the unused dimensions are said to 1.

(Refer Slide Time: 03:37)



Let us see some example so suppose I am trying to define such a 3 dimensional packing of the grid of threads. So this is how I will like to go about it so when you launch a kernel so this is launch example of our vecAdd kernel the vector additional kernel. So if you remember we are supplying these 2 parameters these parameters are corresponding to the number of blocks in your grid and the number of thread per block.

Since these are individually all 3D packing's so both of this X and Y correspond to dim 3 data types that there is a structure that we have discussed earlier that we use this structure type dim3. So you define it like this that X is a packing of blocks where you have this many number of blocks in the X dimension and you do not proceed in the Y and Z dimension so these are set as once fine.

So essentially you are saying that okay you are given some n the total number of elements to add for and then you divide it into blocks of size 256. So overall you will have ceiling of n / 256.0 I hope you can see this and so you are essentially trying to define these many number of blocks

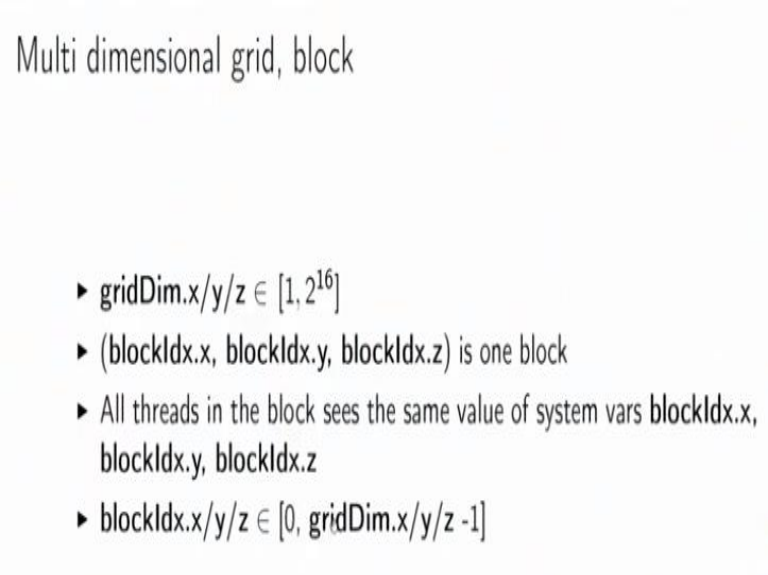
and these 256 is here because essentially you are defining a block as a collection of 256 threads again you are not going to multiple dimensions while you are defining blocks.

So the blocks are also defined by this type Y sorry the variable Y whose type is again dim3 and this Y is again also defined in the single X dimension that it is going to contain 256 threads. So overall you have n number of threads is divided into n/256 number of blocks inside each block you are going to have 256 threads. This is the arrangement that I am choosing here right so with this definition of X and Y when you are launching at VecAdd kernel what is happening is you are passing this X and Y as launch parameters for the kernel.

So when the kernel is launched the threads get packed into I would say tightly neat blocks of 256 size and in total this n / 256 number of blocks are getting launched. I can write this formally like this because this is the usual way you will define it every CUDA kernel we will expert as launch parameters 2 elements here each of this elements dim3 tuples. However since this is the simplistic single dimension kernel I can also directly write like this. So this comma actually separates the number of blocks and the number of threads per block definitions.

So since I have only two entries the CUDA compiler would be smart enough to identify okay will then I am really talking about a single dimensional grid containing blocks only in the X dimensional indexed in the X dimension and again inside the block also I do not have the multiple dimensions really it will automatically assume these kind of definitions as discussed earlier here. And so it will also think that inside each block I have got all the threads packed in each dimension fine.

(Refer Slide Time: 07:13)



Now this is the most important part I believe in the earlier program example simple program we have already discussed that you have a simple CUDA program there are certain variables like thread id's and block id's which get automatically defined and every thread as got its own local value of the thread id and block id will get back to that again from the point of view of all multi-dimensional build and block here.

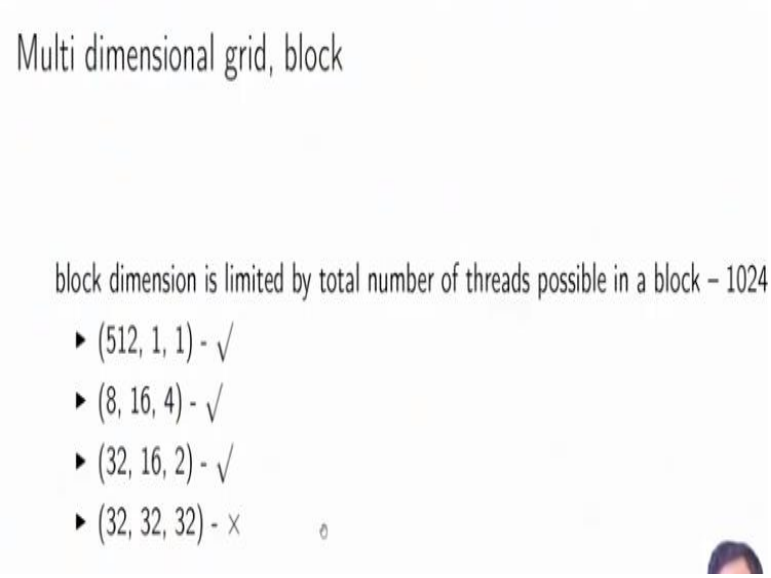
So in general when we are defining a 3 dimensional grid containing the block indices in 3 dimensions essentially the system runtime system of CUDA would automatically define this variable. So these are essentially system variables that the program can always access you do not need to define them or you cannot actually define or initialize them. So the moment you launch the kernel with suitable launch parameters you have automatically available values of this variables that okay for this CUDA kernel what is the grid dimension in the X axis, Y axis and Z axis.

These values can range from 1 to 2 raised to the power 16 okay so that is the maximum dimension of the grid allowed in each of the 3 axis okay. So for each block we will have this 3 block id variables providing me that blocks actual id in the 3D's collection of 3D packing of blocks that would mean for every individual CUDA thread I would have a collection of this 3 variables that mean I will have their values available that the thread can transparently see those values of what is the block id in the x dimension y dimension and the z dimension.

And of course for all the threads sitting inside 1 block these values should be same that means for all the threads inside 1 block they can all access these block id x, block id y and block id z and these combination of value should be exactly same. Because they are technically inside the same block so just to summarize that all the threads in the same block would see the same value of this system variable block id extra text block id x dot y block id x dot z.

And again you can understand definitely that these id x variable values should be varying inside this range of grid dimension right. Why because it is obvious because you are essentially defining a grid which is constituting a 3D packing of blocks the grid is the 3D packing of blocks and each dimension the grid dimension is ranging from 1 starting to 2 to power 16 right so every block is definitely in the x direction or the y direction or the z direction whichever way you go. The blocks id as to be less than the grids allowed dimension in that x, y or z axis right.

(Refer Slide Time: 10:33)



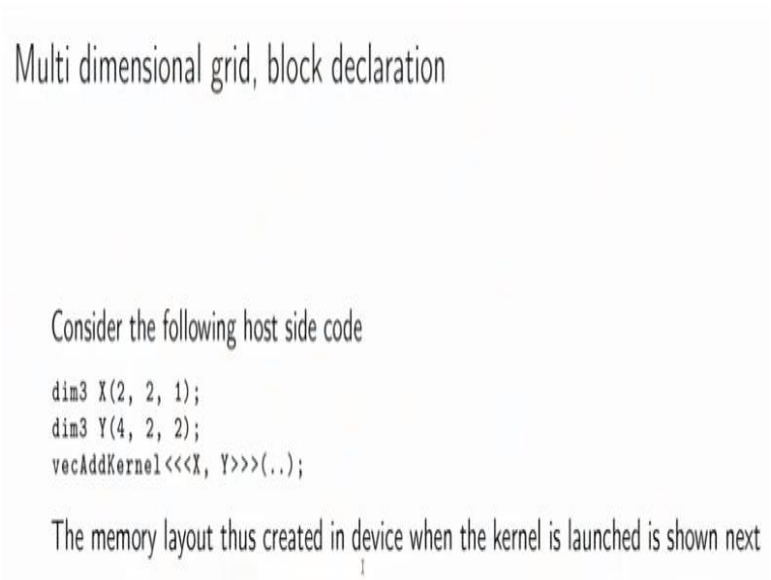
Now there is some other significantly important thing here which is that a blocks dimension is limited by the total number of threads that you are allowed to pack inside a block and this number is 1024. So when you are launching a CUDA kernel the blocks dimension sorry the blocks total number of threads packed into the block cannot be more than 1024. Now of course this is a number that may vary with architectural families which may be in future significantly higher compute capable architectures of the GPU's this number may change this is something which comes as a restriction based on the architectural content the maximum register file size possible and all that.

So right now we can assume that a block has a next size that means there is a upper bound on the total number of threads that you can pack inside the block and it is 1024. So that automatically restricts your possible block ID your possible definitions of a block right. Because as you can see we have given here is an example 4 possible block definitions so in the first definition we are saying that ok let us pack all the threads in the x dimension.

So you are defining a block of size 512 all the threads are in the x dimension so that is perfectly fine because the total number of threads is less than 1024 same thing for the next 2 definitions however if you look at the fourth definition we are saying that let us define a block where you have the threads 32 threads in the x dimension you are letting the thread id x dot x variable go from 0 to 31 and similarly thread id x dot y 0 to 31 and thread id x dot z from 0 to 31.

So if we do that then the total number that is 32 cross 32 cross 32 the number of threads is of course greater than 104 of course this is not allowed.

(Refer Slide Time: 12:48)



So as further example of grid and block declarations let us consider this following piece of host code now why again do I bring in this id of another host code because of course we had another example well this was the much simpler example because technically we had the packing of blocks all in the 1 dimension and inside the block the packing of threads also in the 1 dimension.

Hence just to visit a more complex example let us look at this so here we are saying that okay let us consider grid definition where the blocks are packed in 2 dimensions. So in x dimension you have 2 blocks and 1 dimension you have 2 blocks and also you have inside the block you have a 3D arrangement this threads so you have another dim3 type y with these arrangement 422 and overall you are making a vecAdd kernel call with launch parameter x and y.

(Refer Slide Time: 13:55)

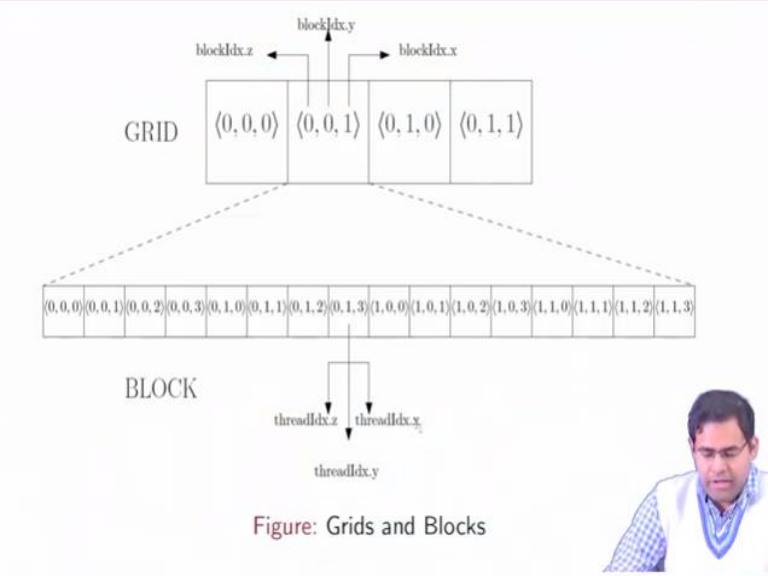


Figure: Grids and Blocks

So this would create a memory layout in that device and the kernel when it is launched the corresponding memory layout for that we have a picture here now this is something which is very important we are just trying understand through this picture that what is the impact of this kind of definition of the packing of threads on the variation of the thread id and block id indices like that.

So since I have defined the grid here as a 2D arrangement so we have as you can see this block id z variable always 0 right. So that is why this is the like more of a hierarchical figure so on the top I just show the arrangement of blocks right. As you can see we have defined the number of the blocks collection as 2 to 1 x so the first entry is corresponding to the x dimension the entry corresponds to the y dimension and the third entry corresponds to the z dimension.

So coming here the way the indexes are arranged here is the opposite so at the very first MSB I would say most significant position I mean the LSB the least significant at the extreme right I have the x dimensions encoding which is increasing and then I have the y dimension encoding

and then I have the same dimensions encoding. Since z is set as 1 here that means we are going to increase the z index.

So it is always 0 of course the index calculation starts from 0 and they go up to the next dimension in the x or y space -1 right which would mean here for this definition my thread indexes in the x dimension are going to vary between 0 to 1 y dimension 0 to 1 and z dimension has to be fixed at 0 right. So that is what we have here all the z dimension encodings are set to 0 right.

The x dimension encoding they vary between 0 and 1 I guess you can see 0, 1 again 0 again 1 and as you can see that the y dimension where encoding is also varying from 0 and 1. But if we were trying to show the arrangement here we are increasing in x first and then we are saying that let us increase in y and we are saying that okay the blocks are arranged like this one after the another right.

But now we need to look deeper right so this is how we will have the block id x variables varying for each of the blocks. So we are having 4 blocks and this are the 4 possible triples or collections of block id variables that we will have for each of these 4 blocks. Now what about the situation inside the block of course inside the block I am going to have the threads and I am going to have 4 times, 2 times 2 that is 16 threads right.

How are the threads going to be indexed right so as we can see that this is again a packing in the 3D space and you are going to have thread ID x dot x variable to range from 0 to 3 thread ID x dot y to range from 0 to 1 and thread id x dot z again going to range from 0 to 1 right and that is how we have it here right. So we start with the thread with indexes all 0 and then I have the thread id x dot x increasing to 1, 2, 3 again we will start from 0 the thread id x dot y increases from 0 to 1 right.

So I have again 4 consecutive locations where it same the encoding at the x position as the first 4 but in the y position we instead of having a 0 here we are having a 1 here right and then again this thing continues this similar pattern would continue is just like counting over multiple dimensions instead of now having 0 in the z dimension you just start having 1 in the z dimension and repeat the scheme the again.

So then considering 1 block inside that block all the threads would be having index triplets arrange like this if you specifically pick up one single thread for example let us pick up this thread so these are threads for which as you can see that the system variables would be assuming values like this. So the thread id x dot x variable would assume the value 3 thread id x dot y variable would assume the value 1 thread id x dot z variable would assume the value 0.

For this thread the block id variables would assume values from here similarly I would have this arrangement repeating for the next block again this arrangement repeating for next block. So if I pick up some thread from this block it would have a specific collection of thread id x y and z right and for that the block id variables would have this 3 values right. I hope this is clear that how each thread can figure out can see what is the corresponding system variable values for it.

That means if you are going to write a CUDA program you can actual compute that what is the actual position of single thread among this c of threads by computing a suitable expression mage using this variables of thread id x y and z and block id x dot x dot y and dot z right. Because for every thread this collections of all this 6 parameters is going to be unique. It may happen that i have this triplet of thread id x dot x dot y and dot z same for 2 different threads in the entire kernel.

But of course they have to be from different blocks so all though the thread id x collections are same the block id x collections have to be different. So in that way I can say that for every thread this collection of the overall 6 parameters we will always be unique.

(Refer Slide Time: 20:40)

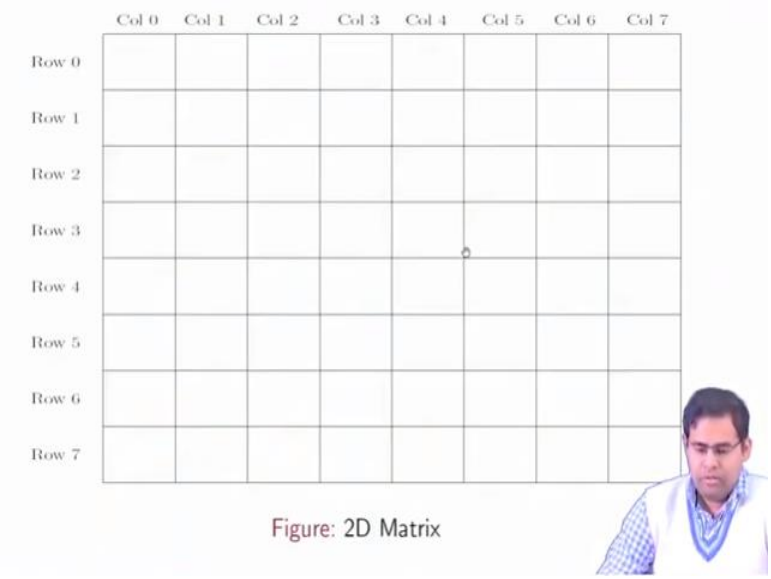
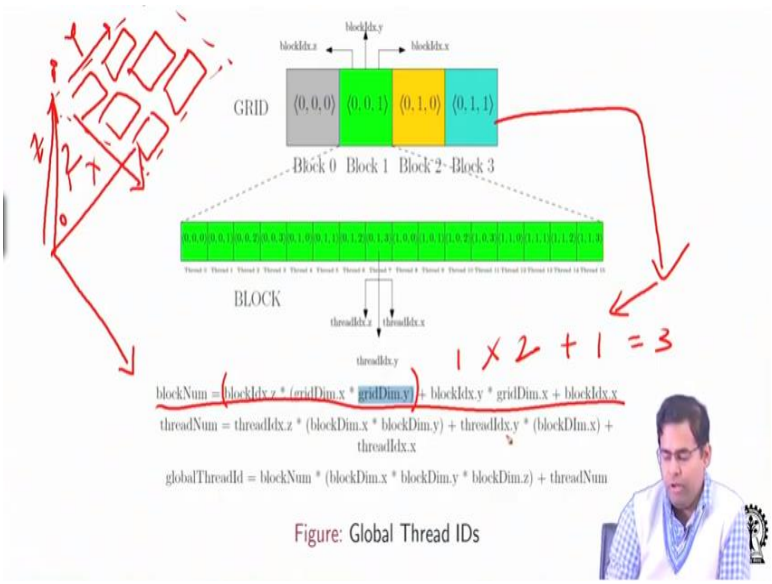


Figure: 2D Matrix

Now let us take an example here that so how really is this definition of threads and blocks going to help me suppose I am trying to use this packing of threads to define a 2D matrix. So I have an 8 cross 8 matrix right and I use that packing of threads to define the matrix how does that work well as we can see that I am actually having blocks of size 16 here and overall I have 4 blocks so that total number of threads that can be packed inside this grids of blocks is 4 times 16 which is 64.

And here I am trying to define a 2D matrix which is in total going to have 8 cross 8 that is 64 elements. So I can use this arrangement of threads for storing all the values of the matrix or me be computing the value of the matrix. So let us say see that how it is indexes can be derived for each thread.

(Refer Slide Time: 22:05)



So first of all representation purpose we just color coat this blocks in different colors this 4 blocks in different colors and we just focus on the green block here and see that okay for one of the specific threads here the thread id x parameters are 013 right and block id x parameters are also 0, 0, 1. Now just to understand inside this entire collection of the locations of the 2D matrix how can this thread figure out exactly what location does it correspond to.

So for that you can see that how this different parameters or the system variables get linked up and ho those parameters can be used to compute a position of the thread with respect to the overall collection of all threads. Now for doing that we observe that these are few relations that are always going to be true. So what is that so what is the current block number for a given thread.

So I can find that out by evaluating this kind of expression right that the okay for this thread the block number is given by the block id in z dimension multiplied the dimension of the grid in x and y dimension because okay when I am talking about the block id in z dimension that means lets the value be q that means I have already come across a collection of these many that is grid dimension x times the dimensions y number of blocks for 2 values of block id x dot z right.

After that you will have to cover whatever is there in the y dimension so that is we will do a block id x dot y times grid dimension dot x and finally after that we have to enumerate ok what is the remaining number of block id x dot x and that would give you the exact number of the block.

Si I hope this is clear like we are trying to figure okay given this parameters how do I find out in which block I am right now positioned how can each thread find out its corresponding block number out of this blocks.

So in this representative example as you can see you have 4 blocks for this specific thread which is in block 1 how can I figure out its block number that the block number if indeed 1. How can we do that? So you can apply this system of equations and try to evaluate so what do you really get here so you are going to get that okay the block dimension in the z access is 0 so this part does not work the block dimension in the y axis is also 0.

So this part of the expression also does not come into play so you are only concerned about block id x dot x and that gives you 1. Similarly for example suppose I am talking about block 3 and we are talking about 1 thread in block 3 and how does it figure out that okay it is indeed in block 3. So for this thread the block id x dot z is 0 so this part of the expression cancels out right block id x dot y is 1 so that would get multiplied by the dimension of the grid in the x axis which is 2 right now why is that.

Now let us just review back so as you can see the dimension of the grid in the x axis is 2 and on the y axis is also 2 right. So based on this since this is 1 so and this is 0 so this part is completely cancelling out and the next part for that I get the block id as 1 gets multiplied by the grid dimensional which is 2 plus the block id x dot x that is 1. So overall you have 2 into 1 + 1 that is 3 so just to work it out here.

So we will just see that if I am speaking of this block this value comes as 1 times the grid dimension 2 plus this that would be 3 right. So just to explain this expression again for example in here we do not have really a case for the first part how would that have come suppose I have got the grid in the actual 3D space right. So then you would really be counting first collection of blocks in the z dimension so you will have some number of blocks in the x dimension and in that way.

So suppose in the z dimension you are sitting at position 2 that would mean you have already come across this kind of 0, 1 and 2 layers of blocks right. So your block id of in the z dimension would get multiplied by these many blocks right because this is packing in the x direction and

the y direction so that is why the block id x z will get multiplied by the grid dimension in x and y axis and then you come to your current dimension position in the z axis and then you count the number of blocks that are there in the y dimension you multiply it by x.

So that exhaust your count by all the blocks located in the x dimension for each block id y value and then finally you would add the block id x.

(Refer Slide Time: 28:46)

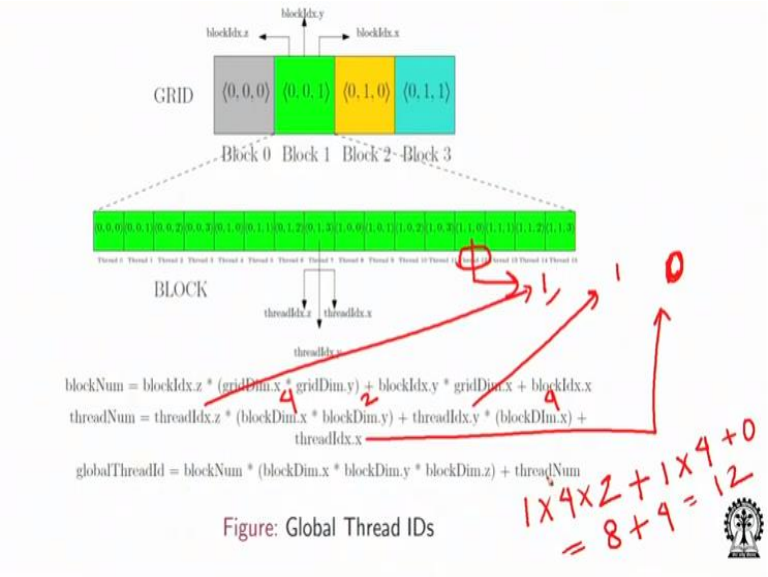


Figure: Global Thread IDs

So we will see that it would get better with in terms of examples once we see some examples code here. Now in that way we can compute the number of the block in the given thread how do I really compute the number the exact number of the thread or the global id of the thread with respect to this entire arrangements. So the thread number would be given by the z index of the thread multiplied by the dimension of the block in the x and y axis.

Because actually when I am talking about z index of the thread then I have exhausted the previous blocks that this many blocks this block dimension times block dimension y number of blocks right times this thread id z and then you have to compute this term which is thread id x y times the block dimension x and then you have to just add the dimension value in this dimension. Let us take an example to make things much more clear here.

So for example if you consider this thread so the thread id x dot z value is 1 and so basically this is your thread id x dot z and then your thread id x dot y value is again 1 and your thread id x dot

z value is again 1 sorry this is 0 and the block dimension in the x axis as you can see for here is 2 block dimension in the y axis is also 2 right and this value is also 2. So overall what do you get is 1 times 2 times 2 + so that is 6.

Now if you so if you look at the position here so we are considering this 1, 1 and 0 so this 1 gets multiplied with the block dimension I believe the block dimensions let me just check that once sorry so in the x dimension is 4 and then y is 2 and z is 2 sorry so here we just replace this entity by a 4 and also just check this is also 4 so in x dimension is 4 y dimension is 2 so now these are fine so this is the thread id x in the z dimension in the thread id x in the y direction and here again the block dimension is 4 so that gives you 8 + 4 that is 12 and as you can see here the thread id is indeed 12 right.

Sorry for this delay here so overall we can see that if you are given the proper thread indexes then the system variables values get automatically initialized and each thread can compute its block dimension and the thread dimension figures and then you can figure what are the block id (()) (33:44) values it an compute which block it is in and inside the block which thread it is in. For example if I pick up this threads corresponding block dimension since there 0, 0 and 1 it will easily apply this formula only the last term would be useful it can figure out that the block number is 1.

I think for block number we did not have a big example here the block dimensions really small in value but just map this idea of thread number computation here and you can see really see how the block number computed for a more complex example here right. And the important thing would be that okay a thread number can be computed for a position of a thread inside this block and a thread can also find out what is the number of block in which it is position using these two values.

The thread can find its exact position in the global thread id variable that is its exact position among all the thread have been launched which is nothing but the global thread id is equals to the number of blocks that have been covered to block 1. So in each block you have got how many threads block dimension x times y times z these many threads plus the thread position in the current block this gives the threads global id which we call as the global thread id.

So this is the exact position of the thread or exact index value of the thread among all the thread that have been launched by the kernel. Well I think with this we should be ending the lecture here and thank you we will explain more on this in the next lecture thanks.