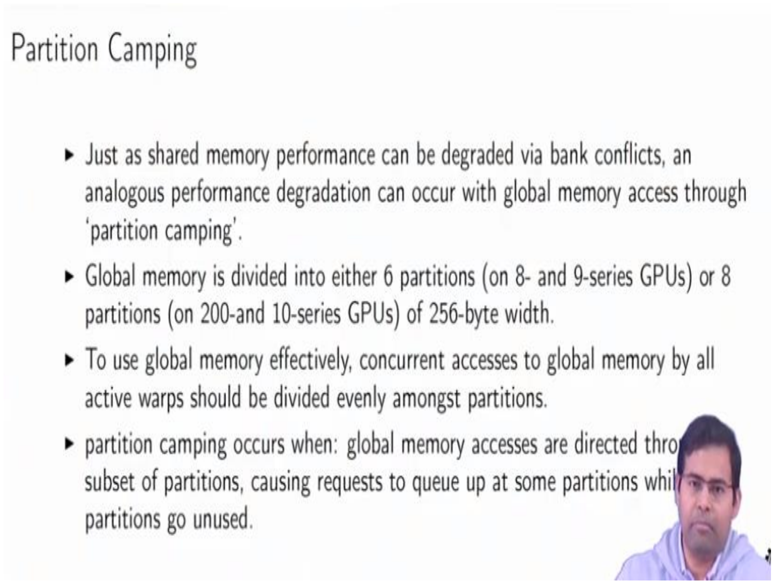
GPU Architecture and Programming

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

Module No # 06
Lecture No # 27
Memory Access Coalescing (Contd.)

Hi welcome to the lecture series on GPU architectures and programming so in the last lecture we have seen certain examples of how coalesced access on the shared memory was helping us to do nice I mean nice optimization with respect we are computing transpose of a matrix. So we figured out that with the possible optimization that could be done we were able to achieve almost similar execution times which shared memory based transpose computation.

(Refer Slide Time: 01:01)



Partition Camping

- Just as shared memory performance can be degraded via bank conflicts, an analogous performance degradation can occur with global memory access through 'partition camping'.
- Global memory is divided into either 6 partitions (on 8- and 9-series GPUs) or 8 partitions (on 200-and 10-series GPUs) of 256-byte width.
- To use global memory effectively, concurrent accesses to global memory by all active warps should be divided evenly amongst partitions.
- partition camping occurs when: global memory accesses are directed thro subset of partitions, causing requests to queue up at some partitions whil partitions go unused.

As is the case which shared memory based simple copy operation. So this also highlights the architectural issues that one has to keep in mind while writing programs for the GPU architectures. Now upto this point of time the memory conflicts that we handle where of 2 types once was how to coalesce global memory accesses and how to remove bank conflicts while accessing the shared memory.

Now there is a related problem with respect to global memory which is also relevant in this concept in this context and it is known as problem of partition camping. So this is the again something related to global memory and we thought it would be good to discuss this after we discuss this shared memory bank conflict because the idea is similar in that sense. So just like we have this notion of bank conflicts in shared memory the reason being this shared memory is decided across banks in order to facilitate parallel access.

Similar ideas hold for global memory that the global memory is also divided into certain partitions. So depending on the GPU series there can be the different number of partitions so it can be either a 6 partition I mean a 6 way partitioning of the global memory or it can be 8 way partitioning of the global memory. So we will consider the 8 way partitioning where for each partition we consider that the memory is 256 bytes wide that means I can read a chunk of 256 bytes from 1 partition in the global memory.

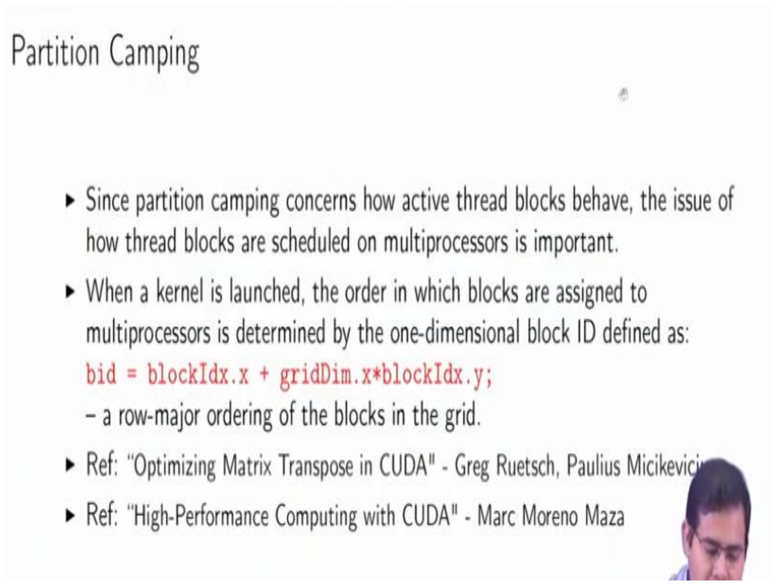
So that would also mean that since I have let us if I consider an 8 way partition global memory I can read 8 chunks of 256 bytes in parallel from the global memory. So what we like to do is to access the global memory effectively and facilitate as much concurrent access as possible who get data from the different partitions in the global memory in parallel right. Now why I mean just like we have this issue of back conflicts we have the similar issue here known as I mean the name is different is called partition camping so why will that occur.

It occurs with the global access memory are direct to a subset of partition just like imagine multiple threads in a warp they are executing and they are trying to access the same bank of a shared memory. Now just take the idea at a higher level consider that you have different SM’s and blocks executing in the different SM’s and they are trying to access in parallel the same global memory partition that would actually queue the corresponding access request and the access will slow as a result of that.

So just instead of having instead of considering access of threads inside the warp for the single bank of shared memory you go to the higher level I have repeating here that you consider a multiple blocks which are executing in different SM’s and as we know that each SM as a memory controller based interface with the global memory and each of this blocks are placing their request to their respective memory controllers for accessing the global memory and it happens to the case where those request are going for the same partition.

Now inside the same partition I can only read in 1 transaction to 256 bytes right so all those queries we get q dot and it will create this issue of partition campaign.

(Refer Slide Time: 04:47)



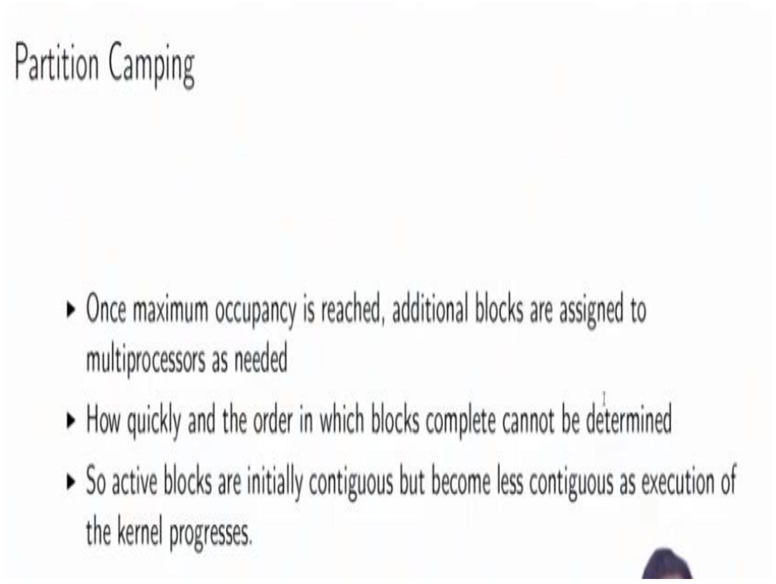
Partition Camping

- Since partition camping concerns how active thread blocks behave, the issue of how thread blocks are scheduled on multiprocessors is important.
- When a kernel is launched, the order in which blocks are assigned to multiprocessors is determined by the one-dimensional block ID defined as:
  `bid = blockIdx.x + gridDim.x*blockIdx.y;`
  – a row-major ordering of the blocks in the grid.
- Ref: "Optimizing Matrix Transpose in CUDA" - Greg Ruetsch, Paulius Micikevici
- Ref: "High-Performance Computing with CUDA" - Marc Moreno Maza

So I mean we will summarize like this that since the partition camping concerns how active thread blocks behave the issue of how thread blocks are scheduled on the multi processors is important. As we are discussing that the partition can be accessed in parallel by this SM's so what really matters is how the blocks has been distributed across the SM's. Now how does it get done? So when a kernel is launched the order in which blocks are assigned to a multiprocessors is determined by the single dimensional block id.

So suppose you have a 2D block so if you map this arrangement of 2D blocks into a row measure ordering of blocks. So then you get a unique block id right following this relation I mean so essentially we are linearizing the 2D arrangement of blocks to get a total order on a block of id's and this single integer total order will actually tell you how the blocks are getting distributed. So the following are (()) (05:55) arrangements the blocks will get distributed following this BID value right.

(Refer Slide Time: 06:01)



Partition Camping

- Once maximum occupancy is reached, additional blocks are assigned to multiprocessors as needed
- How quickly and the order in which blocks complete cannot be determined
- So active blocks are initially contiguous but become less contiguous as execution of the kernel progresses.

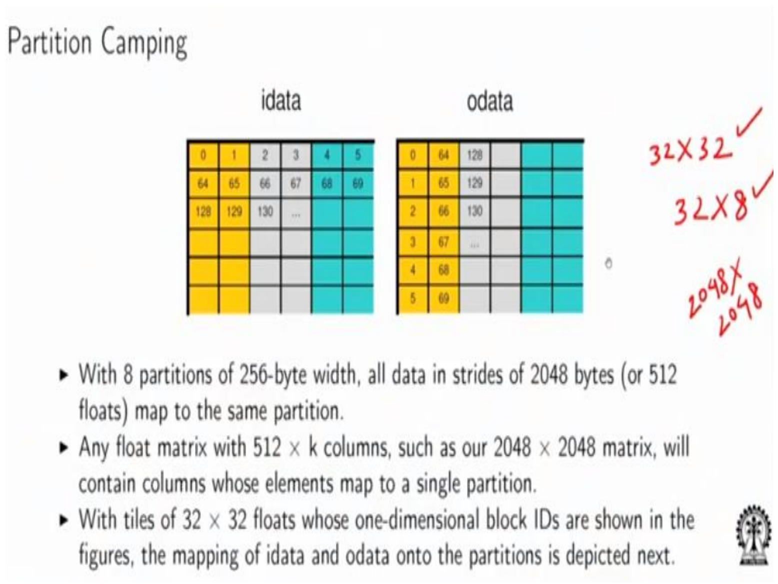
No so in this way once blocks gets distributed across the SM's technically I have then single SM considering as SM I mean doing our data crunching over significant size data so that all the SM's have got fully full engagements and each SM have been mapped with multiple blocks. So when I have reached this maximum occupancy I mean addition I mean additional blocks are assigned to the multiprocessor as it is needed.

Now suppose I have this assignment of block done but then there is no control so let us assume that I have a set of blocks as assigned to multiprocessor 1 set of blocks assignment set of multiprocessor 2 set of blocks into multiprocessor 3 so and hence so forth but then as we are discussed earlier there is absolutely no control under execution ordering of the blocks across the SM's right. Because the high level scheduler of GPU system is distributing the blocks across the SM's inside the SM's you have a low level scheduler which is deciding which is actually forming the warps from the blocks and it is dispatching the warps to execute across a SP codes right.

So you do not have really any control over in what the blocks progress right as we have discussed earlier inside an SM's there will be a warp the low level schedule of the warp scheduler who will follow some method it will dispatch warps by dissolving dependences but across SM's there is no such control and it is done intentionally to achieve maximum parallelism as much as can be extracted from the application.

So how quickly and in which order the blocks get executed cannot be determined but the active blocks are initially contiguous I mean the active blocks that are initially contiguous but become less continuous of the execution of the kernel will progress that is what are packed.

(Refer Slide Time: 08:20)



Partition Camping

- With 8 partitions of 256-byte width, all data in strides of 2048 bytes (or 512 floats) map to the same partition.
- Any float matrix with 512 × k columns, such as our 2048 × 2048 matrix, will contain columns whose elements map to a single partition.
- With tiles of 32 × 32 floats whose one-dimensional block IDs are shown in the figures, the mapping of idata and odata onto the partitions is depicted next.

But now let us try and understand that how the blocks will be a block across the SM’s are going to access the data in the global memory. So for this problem we consider a situation that so for this problem we consider this kind of a definition of threads, tiles and blocks so if you remember from our earlier examples we are considering that the data tile size is 32 cross 32. And we have blocks of size 32 cross 8 right so it is a 2D arrangement here.

Now with this kind of a setting so and also we consider that with this kind of tile size for the data and with this kind of dimension of blocks we are trying to do a transpose kind of computation on a matrix whose size is 2048 cross 2048 okay. So if I am doing it like that we let us try and identify how data maps on the global memory. So since in the global memory i have this division of 8 partitions let me just number the partitions 0, 1, 2, 3 like that.

Each of the partitions at 256 bytes wide right so sorry this is not the numbering of the partitions the partitions have shown here using the colors and we will define the numbering here. So the first yellow color part is basically partition 0 the second one is partition 1 and like that right. Now since each of this partitions is 256 bytes wide so if I have data sitting in strides of 2048 bytes or for that matter 512 floats because 512 float means for 4 byte so multiply and get 512 times 4 is 2048.

So if I have 8 partitions each partitions is 256 byte wide so if I just do this calculation that this n multiplies to be 2048. So I have the large sequence of data points. So every data points and whatever is its partition that data point index + 2048 bytes would give me so I mean that data starting from that data point to the data point which is sitting 2048 bytes I mean further from that they both will belong to the same partition right. I mean is really obvious because each partition is again 256 byte wide.

So you get this stride size of 2048 just by doing this simple calculation of the total width offer by the 8 partitions. Now what does this really mean in terms of the data types for example if I consider a float matrix so essentially a float matrix let us say it is size is 512 cross K right so I have 512 rows so 512 rows means the row is 512 times 4 bytes so that is 2048 bytes wide. So essentially each column will get mapped to a unit partition right to a single partition.

So if I have a 2048 cross 2048 matrix here so essentially its 2048 is a multiple of 512 right so again I will explain what is going on. I have 8 partitions each partition is 256 byte width so overall width is 2048 bytes. So I can accommodate 512 floats into 2048 bytes so if I consider a float matrix with 512 cross K number of columns then essentially what is happening every column of data maps to a single partition right.

Now the logic will hold for a matrix of higher dimension where the dimension is a multiple of 512 right now since 2048 is also a multiple of 512 so for our target matrix of this size the columns will also get mapped to the single partition here right. Now since I am considering data tiles of 32 cross 32 floats they are accessed by 1 dimensional blocks right and so we are considering our earlier examples right where if you go to the block dimensional definition it was 32 cross 8 so we are considering that I have these many threads to process the tile of this size right.

Now if I do a mapping of the tile with respect to the block id right then I try and figure out how exactly this tiles are getting distributed across the partitions so with tiles of 32 cross 32 floors whose 1 dimensional block id's are shown in this figure again I will just repeat the blocks are smaller because inside the block we have set of threads and each thread accesses 4 elements in the tile but since 1 block takes care of accessing 32 cross 32 floats I represent that tile size with the corresponding block id why?

Because the number of threads in the block does not matter in this context I am trying to see what is the memory consumption by that block. So that block of small size the block of id 0 or 1 each of them are working on a tile size of area 32 cross 32. Now since this are 2D map so this 32 cross 32 tile let us understand how they are distributed across partitions so in the x direction I have 32 data points.

And so each of them are of size 4 bytes right so I have 128 bytes in the x direction for each tile and since the partition is 256 bytes wide I will just repeat this part again. Partition is 256 byte wide each tile in the x direction it holds 32 floats that would mean it is 32 cross 4 bytes so it is 128 byte wide right that means I can accommodate 2 tiles 2 consecutive tiles in 1 partition so there is a 2 level argument here.

First one I would say is that why I am mapping this tile with a block id I am just writing 0 for tile 0 because I am seeing that look at this tile of 32 cross 32 will be processed by a corresponding block of size 32 cross 8 but what is the way in which the tile is going to be represented here. So the tile is in 2 dimension so we it maps to the memory in the x direction I have 32 floats being arrangement so they would consume 128 bytes.

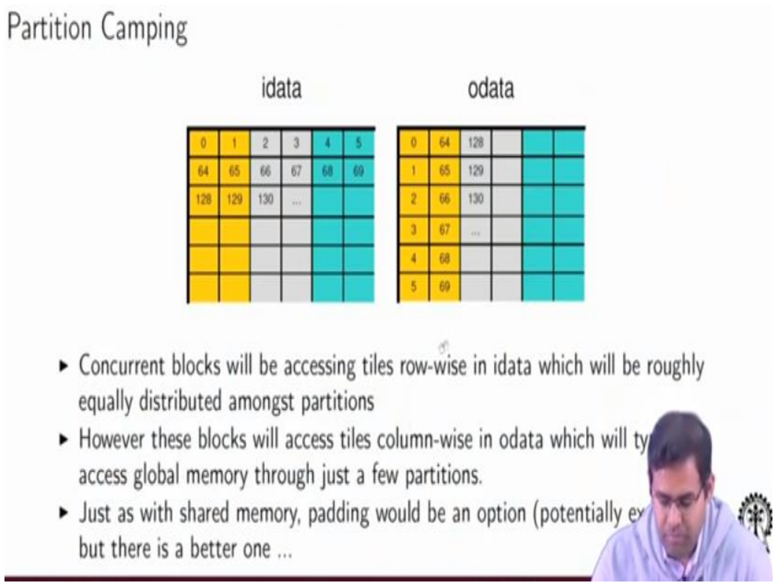
So I can have 2 consecutives tiles mapped in this memory right so in this way my input data that is the high data matrix considering it the very big matrix is getting map like this right. So how many tiles do I really have in the x direction of course so that would mean you decide 2048 / 32 you have got 64 tiles right. So the tiles with id 0, 1, 2 like that I mean I am writing the corresponding block id’s here right when the block id which are going to work in this tiles so you map this tiles in the across the partitions 0 to 63 right.

Now so overall I have got 8 partitions now so in each partition I have got 2 of them sitting 01 and then 64, 65 and like that. So this is how the original arrangement is here right so this is how it is going to continue but the operation that I want to do on this data that is in the global memory is to perform a transpose operation right. So the expected pattern in which the output data should be organized as to be like this as we can see just take a look into the o data part.

So again just to summarize the overall problems space here I have the input and the output matrixes here and they are arranged in idata and odata. I am just trying to show how they map across partitions so overall I am just figuring out that inside each partition I can have 2 consecutive tiles here right and in that way if I continue I will get this kind of a distribution right. So I have overall this 8 partitions and all data which is in strides of 2048 types will just keep on repeating and for each tiles I have this 0 and 1.

And here I am trying to show 1 possible arrangements of course this is not this indexes are just trying to represent that how the 1 dimensional block id's get mapped here right. Now let us try and figure out that for doing a transpose computation what is the corresponding overhead in terms of partition camping from the global memory side.

(Refer Slide Time: 19:34)



Partition Camping

0	1	2	3	4	5
64	65	66	67	68	69
128	129	130	...		

0	64	128			
1	65	129			
2	66	130			
3	67	...			
4	68				
5	69				

- Concurrent blocks will be accessing tiles row-wise in idata which will be roughly equally distributed amongst partitions
- However these blocks will access tiles column-wise in odata which will ty access global memory through just a few partitions.
- Just as with shared memory, padding would be an option (potentially ex but there is a better one ...

So earlier we have been looking into the problem that 1 block is working in the shared memory and how much optimizations are getting possible. But now we are taking the global view of the problem right so I have got concurrent blocks executing across SM's and they are accessing the tiles from the global memory. Now let us understand what is the access pattern from the let us understand what is the access pattern such that from the input data matrix and what is the access pattern for the output data matrix right.

So concurrent blocks will be accessing the tiles row wise in the idata matrix and in this so as we can see this roughly be equally distributed among particles right. As we can understand the block id's the concurrent blocks that are executing zeroth block, first block, second block, third block like that each of them have got a working set which is the 32 cross 32 tiles size and 2 consecutive tiles map into the same partition right.

So since it is the row wise access so I can expect that the concurrent in a average scenario they are consecutive block id's and they are accesses are almost equally distributed among the partitions. So I mean of course after some time they lost the synchronization in terms of which blocks is executing in which SM. But since the blocks are distributed across SM's and there is no fine grain control over there execution order roughly from each SM whatever partition is being accessed for a block overall since the access have to be concurrent.

On the average I would expect that they are roughly equally distributed among the partition again we will just try to understand that in a perfect scenario it will be equally distributed otherwise also it will be roughly equally distributed the reason because I do not have concurrency control across the SM's I do not have control over the block execution ordering across SM's.

We are expecting that concurrent blocks that are executing across this SM's they are accesses to the partitions that are equally distributed just because I am con-current blocks will access that tile data row wise right. Now if I consider how this concurrent across SM's are going to access the output data again I do not control over concurrent blocks across the SM's but they will access the data column wise in odata.

Now let us understand what is happening here so I have got blocks from different multiple processors which are going to access the odata and as we understand since the blocks are roughly distributed equally across the SM's. So they are whatever they fetched row wise they are trying to write column wise in the overall arrangement of the matrix and that will also get reflected here in terms of the global memory accesses that this blocks will access the tiles column wise in odata by columns here we mean this column of tiles sitting in the same partition.

So I am again I will just summarize I have different blocks executing across SM's they are accessing this tiles in parallel right. But when this blocks are going to access the data in odata matrix for writing it out the writes are going to happen in this order why? Because as we can see that this are the consecutive tile id's and they are sitting in the same column of I mean here column does not mean column of data but the column of tiles indexed with the column of tiles which are indexed with the corresponding block id's be very careful here right.

So since this tiles will be accessed column wise in odata this will typically access only a few partitions again we are repeating here we cannot say strongly that the access will be on exactly the same partition in an idea scenario yes because I have exactly the consecutive block id's which are executing across SM's and they are accessing the same column of tiles in odata which means they are accessing all of them are accessing the exactly same partition.

But since I do not have any control over the execution order of the blocks but still I can expect that their execution order will not be exactly distributed it will be a bit different from the perfect column access but still it is not going to distributed across all partitions but it would be restricted to much smaller number of partitions. So overall the philosophy here is that when you are going to read from idata you are actually reading in a perfect scenario from all partitions in parallel from in a actual scenario from a large number of partitions in parallel.

But when you are writing odata you are in a I mean you may be reading from very small number of partitions in parallel. Now just like I mean shared memory bank conflict you have this issue of partitions camping here because from each partition you can read only in this width of 256 bytes. So essentially you will read he first row here and the first row here in the executive tiles right. Now okay what happens if I consider the optimization with this for resolving the conflict of shared memory well what was the optimization?

The optimization was that okay you just increase the tile dimension by 1 and but did not really use it was a do not care dimension that is what we call as padding the memory with data so that the access I mean access actually resolve the bank conflicts but if we are going to use that same concepts here is going to be potentially expensive why? Because instead of adding a single column you are going to add full tiles here right.

So that is 1 problem but also you I mean since you do not have any control over the blocks that are executing the scheduling. So you are not also sure that how good that approach is going to work in this case we can do it in a different way possibly will see how. So first of all we understand that padding can be an option here but with respect to shared memory in this case the option is much more expensive option.

So when the rights are happening in this order as we can see writes are kind of column the writes are accessing the same partition more frequently I mean writes across blocks y because as we can see that tiles are kind of arranged here in a column wise way right. So that would mean multiple blocks across SM's they are kind of not accessing all the tiles all the partitions available in parallel but they are accessing only a few tiles for few partitions.

Now why would that happen because as we have discussed in ideal scenario it will be the worst case because blocks across SM's suppose they are executing in a perfect last step round robin order then they would be accessing I mean this kind of a single column of tiles so it will be only accessing a single partition but since I am not controlling the execution speed of the blocks across the SM's and their scheduling.

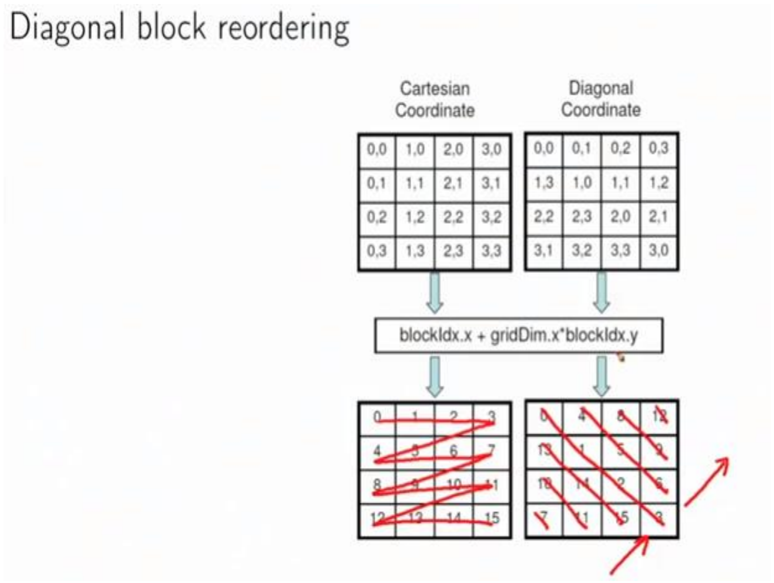
So still I would expect that they are behavior would not be totally uniformed and they would access only a few partitions while doing the writes on the odata. So just to summarize when multiple are going to access the idata it is expected that lot of partitions would get access in parallel but when this multiple blocks are going to access the odata it is expected that lot of blocks are going to I mean only to access a few partitions right.

So in the first case while reading since I am kind of reading row wise data so it is expected that I will read from lot of partitions but when I write only to a few partitions. Now to reduce this issue of partitions camping we can follow the same strategy as was done in case of shared memory. So in that case the strategy was padding by padding we essentially what was did was we introduced a new column in the shared memory.

Now the good thing of introducing that extra column in the tile was that the way in which the data was written earlier the data was getting loaded was that everything from a tile and was thought of conflict but after we did the padding there was full coalescing achieved in terms of both loads and stores from the shared memory. Similar option was would share of course but of course there we can understand it is a potentially very expensive option.

Because you are not just only adding one single column of 0's here right so it is a much more expensive option and it is possible to have a better option here. So let us see what is the better option.

(Refer Slide Time: 30:02)



So in this case what we do is diagonal block reorder now what does that mean so if you look at the normal way in which the blocks are arranged in a memory I mean the blocks are arranged as per the programming model here we follow our normal Cartesian coordinate system right. So following the normal Cartesian coordinate system the block id's in terms of x and y dimension gets distributed like in the left hand side figure left hand side box right.

And this distribution of the Cartesian coordinates they are easily linearized using this formula that has been given and they give me a total ordering of blocks which is the row measure ordering as we can see here is a normal row measuring array. And primarily due to this ordering we get the issue of partition camping with respect to the global memory right because this order is only creating the corresponding arrangement of access right access of the blocks by the respective block id's and you do not want the accesses to happen in such a way and I can actually defer the accesses by modifying the order of accesses by changing the allocation of block id's to the different tiles right.

So essential that is what this picture is about right I am modifying the id of the block who is going to access a specific tile. So just to understand again if you look into this figure I have a mapping in a memory right of the tiles and what we are will trying to say is that the which block of threads is going to access with tile. So the work is same the work is that you to do a transpose operation on the memory what not going to change is what is the block id of the thread block which is going to do the respective warp.

Now why is that going to warp because if I change the ordering of blocks then accordingly the schedule will also pick up blocks following the computation of the block id's right this is the important point I mean so just to summarize here. We understand one thing that the scheduler is going to assign block to SM's following this relation right. So if I am going to change the way block id's are going to fed for computing this things will be different so that is what we are going to do.

So we shift from this Cartesian coordinate system to the diagonal coordinate system so that we can have a different block id computation scheme while doing the access of the memory right. Essentially what I am doing is I am giving a same job to a different block that is what we can say right. Now of course I cannot pass a different block id to the scheduler but instead of that what I can do is I can make the scheduler corresponding block work on a different id memory address.

Now this is the important thing we need to understand here so instead of having the blocks work on this styles so this are essentially the tile number trying to say here that if I am following a one on one mapping of block id's with tile and to respective positions this is the way the tiles are going to the arranged right. Now we are saying that no let it not be the case let the tile indexes there be different because essentially the tile index is represent the way different blocks are going to work on them.

And we modify the way the tiles are being accessed by suitable access expressions. So as you can see from this Cartesian coordinate which shift to something called a diagonal coding so what is that? So we start enumerating blocks in a different way so I am just trying to show how we are going to enumerate the blocks first we will do it here and then we will see how it translates to the original scheme here.

So here this is my access pattern right here I am saying that okay let me numerate them like this primary diagonal followed by the first secondary diagonal in the right hand side. Then this then the other diagonal here then this then here then this so essentially you start with the primary diagonal then you start moving this way and then you switch here and start moving this way and you keep on switching back and forth right.

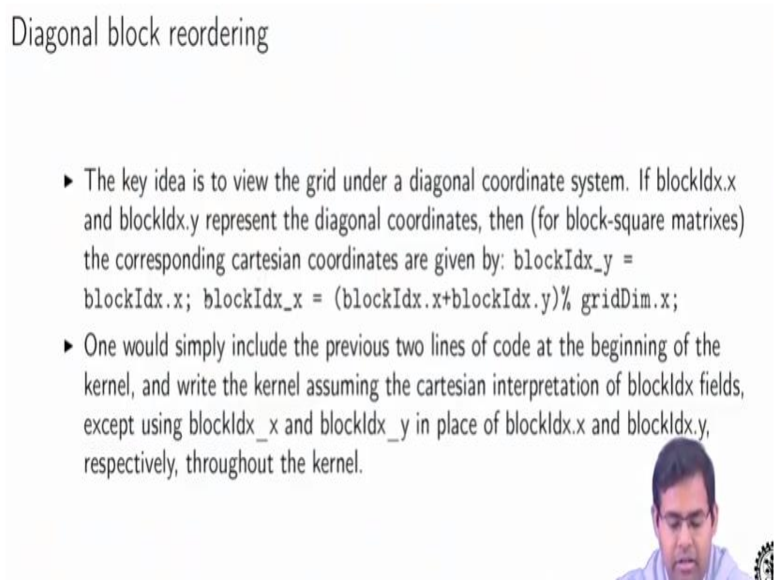
So primary diagonal switch to right side move on diagonal switch to left side move on diagonal so in both cases we are switch you are moving in this direction but you are switching among the sides once in a while right. So I want this enumeration scheme for the ID's right I want this is the way in which the block should be accessing the data right. So for doing that I use this diagonal coordinate scheme so that if this formula is applied on this scheme I get this enumeration order.

Now the question is why do I have to change the scheme can I alter the formula ideally no why because a we are provided with the system own definition of variables in terms of block id's and thread id's right. So what we can really can do is we do not have a direct control or which actually from the we cannot define from our programs given a thread what is its we cannot redefine the block id component or the thread id component values because it is thread id x and block id x they are all system variable.

But what we can do is we can define a different mapping here such that the respective block follows this mapping while accessing the different locations in the memory. So just to understand this was my Cartesian coordinate system which gave me this nice linear representation of block id's I shift to a different diagonal coordinates system on whom if I apply this same relation of block id x plus grid dimension in the x direction times block id x y then I get a different linearization which is more of an access to the diagonal right.

The good this if I have access to the diagonal that ensures that I am writing the output data cross more number of partitions right. And removing the problem of partition camping we will see how but just to now see that okay how is this diagonal coordinate define. So observe any entry here with unequal components so let us this entry and this entry you see the second component and the first component are same and this is true for all the other entries right. So here also the second component here is 1 the first component here is 1 right so they are same.

(Refer Slide Time: 37:46)



Diagonal block reordering

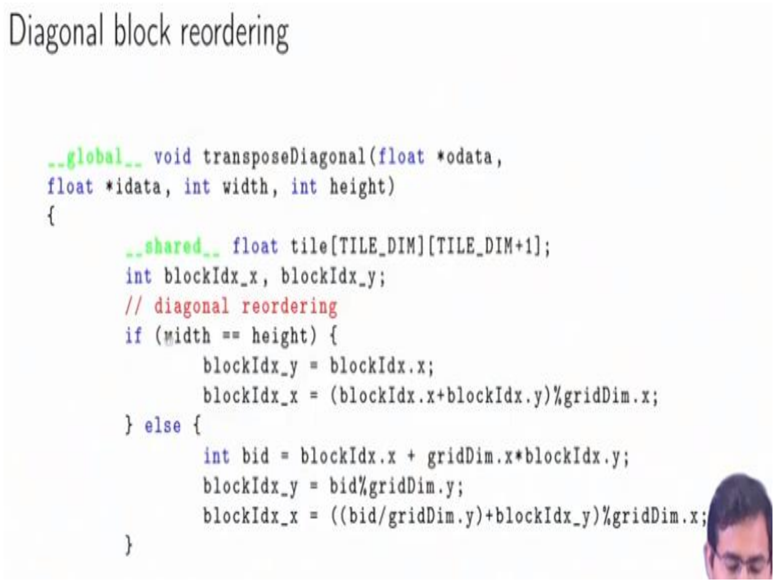
- The key idea is to view the grid under a diagonal coordinate system. If blockIdx.x and blockIdx.y represent the diagonal coordinates, then (for block-square matrixes) the corresponding cartesian coordinates are given by: `blockIdx_y = blockIdx.x; blockIdx_x = (blockIdx.x+blockIdx.y)% gridDim.x;`
- One would simply include the previous two lines of code at the beginning of the kernel, and write the kernel assuming the cartesian interpretation of blockIdx fields, except using blockIdx_x and blockIdx_y in place of blockIdx.x and blockIdx.y, respectively, throughout the kernel.

So what we do is we use this kind of ideas to re-compute the new block id x so original block id x variables are block id x dot x and block id x dot y you define a new diagonal coordinate system and call it block id x underscore y which is block id x dot x right. As you can see the second component here becomes the first component here so this relation holds. And the block id x dot x which is the other variables that means so the second component here is becoming the first component here right.

But what happens to the other component so the other component here which is block id x underscore x is computed using this relation now this is the relation which is easy to check right. So all you need to do is you apply these 2 relations on the original block id x dot x and block id x dot y variables here you will get this modified diagonal coding system that means you get this block id x underscore y and block id x underscore x variables recomputed to give you new values of block id's here right in the x and y direction.

Now why do you really want to use it now you just introduce this 2 new lines of code and use this new interpretation of block id's in x and y direction throughout the kernel. So essentially at the start of your original code yo replace this block id x underscore y you replace the block id x underscore dot y with block id x underscore y's and you replace your block id x underscore dot x with this block id x underscore x values with you are computed here and then throughout your program uniformly you replace the dot x and dot y's with underscore x and underscore y's for the block id's.

(Refer Slide Time: 39:50)



Diagonal block reordering

```
__global__ void transposeDiagonal(float *odata,
float *idata, int width, int height)
{
        __shared__ float tile[TILE_DIM][TILE_DIM+1];
        int blockIdx_x, blockIdx_y;
        // diagonal reordering
        if (width == height) {
                blockIdx_y = blockIdx.x;
                blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
        } else {
                int bid = blockIdx.x + gridDim.x*blockIdx.y;
                blockIdx_y = bid%gridDim.y;
                blockIdx_x = ((bid/gridDim.y)+blockIdx_y)%gridDim.x;
        }
```

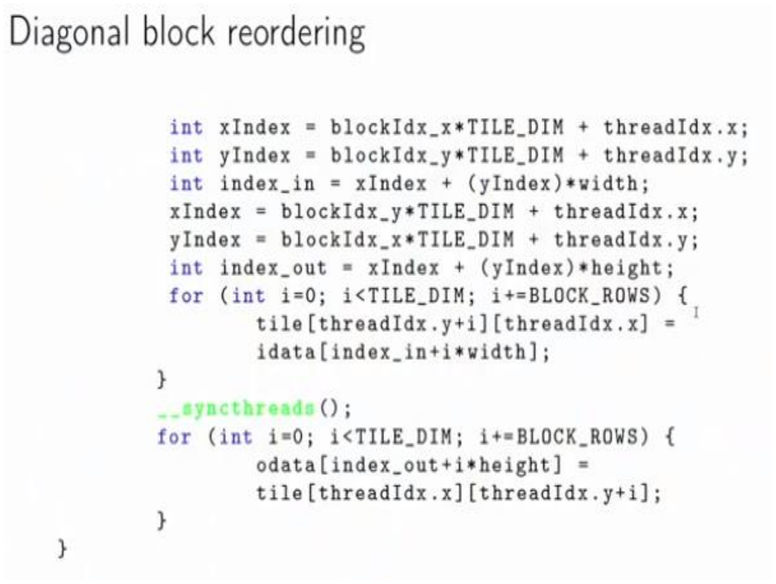
Now so essentially what you do is this is your re-computation right so of course you have to decide about the re-computation in 2 different ways because incase the matrix is symmetric the width is equal to height then this relation will hold uniformly for all block id's and block id x dot x and block id x dot y values otherwise is not symmetric then you have to some extra calculation which is provided here.

I mean this we are not getting details we should be able to understand where looking into the code later on right. So essentially this is the segment of code through which you do the diagonal reordering and essentially you transit from Cartesian code in this system to a diagonal coordinates system and then why because then if you apply this linearization formula then what you get is the different access pattern of the set blocks right.

So essentially what we will do is with this you have a different value of block id x underscore x and block id x underscore y and instead of using the original block id x variable if we use this block id x variables then the order that we just discussed that is the diagonal based ordering that is the ordering in which the thread blocks will be accessing the data. So try and really think what is happening you are not really changing the system variables you are not changing threads schedulers or anything which is it is out of your hand.

But what you are changing is the access expression of each of the blocks right that is the most important thing we need to understand.

(Refer Slide Time: 41:36)



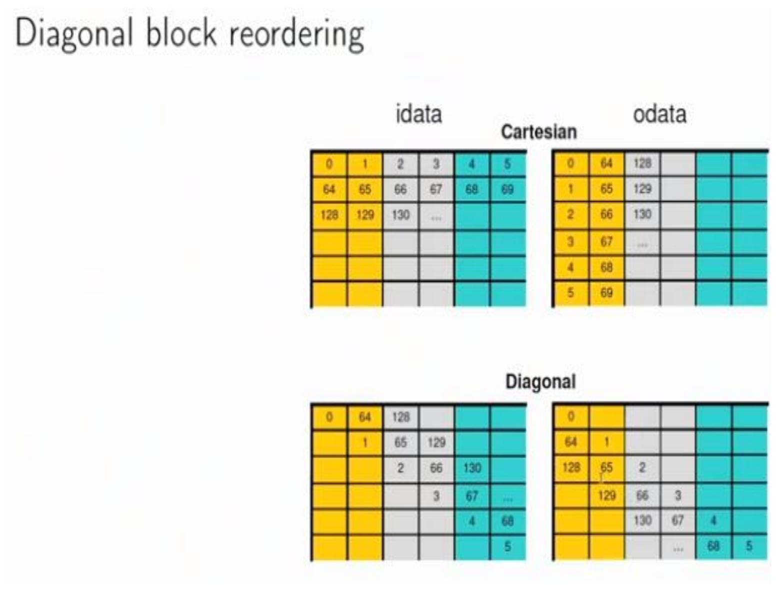
Diagonal block reordering

```
        int xIndex = blockIdx_x*TILE_DIM + threadIdx.x;
        int yIndex = blockIdx_y*TILE_DIM + threadIdx.y;
        int index_in = xIndex + (yIndex)*width;
        xIndex = blockIdx_y*TILE_DIM + threadIdx.x;
        yIndex = blockIdx_x*TILE_DIM + threadIdx.y;
        int index_out = xIndex + (yIndex)*height;
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
                tile[threadIdx.y+i][threadIdx.x] =
                idata[index_in+i*width];
        }
        __syncthreads();
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
                odata[index_out+i*height] =
                tile[threadIdx.x][threadIdx.y+i];
        }
}
```

Without changing anything in the original program you transform the program from one coordinate system to another coordinates system and now when you are using this new underscore x and underscore y variables to compute the x and y indexes to do the tiles based computation that means you load from idata and you again use the tile to write to the odata. Just the original program right only you are using this block id x underscore y and block id x underscore x variables to do the x index and y index computation and then using the x index and y index to compute the index out right.

So this is the only modification that you have right that you just start using underscore x instead of the dot x for the block id's with this essentially what you are ensuring earlier your block id's mapping was exactly the order in which you are accessing the tiles based on the access expression but now due to this alternate mapping you have consecutive block id's but they are making access of the tiles like this right they are going to access the tile value like this.

(Refer Slide Time: 43:02)



So with the same program now let us see how the block id will get distributed when I consider its access of the partitions. So in the Cartesian coordinate system this is what was happening the blocks id’s where distributed in a row wise manner right in the output data came column wise manner. But now if we use the diagonal coordinate system this is the first input data string so you have the first diagonal they came on the right hand side of the next diagonal then the left hand side you will have another diagonal like that.

And then when you transform here what is really happening so essentially the diagonal switch and you get the same effect so all that is happening is the diagonal are switching space right. So for when you are reading or writing that data you have a nice balanced way in which the majority of the partitions are going to be accessed because of this diagonal scheme this is the good thing you get right. So when you read the data you read from majority of partitions because you have the blocks with id’s 0, 1, 2, 3 like that they are going to read from this tiles right.

The blocks are going to read from this tiles essentially the block id represents and what tile they are going to map right and when they are going to write and again to write a following this diagonal based ordering so they are again going to write to multiple partitions in parallel significant number of partitions right. So that is the primary idea here that by making this shift in the coordinate system you are able to increase the number of partitions which are being used for writing the data instead of pure column wise write of data.

And with this we will come to a conclusion of our discussion on memory access policy I hope it was really useful for your increasing understanding of how GPU memory system is organized and how it can be exploited for optimizing your code. Just to summarize I mean remember one thing when we have talked about this block id’s as I have told time and again but I think it is important also.

The block id’s are determining which tile you are accessing right and which tile you are accessing depends on what is its location in the partitions in the memory partitions. Since we have changed the block id’s ordering here the block id’s linearization scheme here I have changed the way in which the partition are getting accessed with respect to read as well as write and in both cases I am accessing a lot of partitions together and that is fundamentally giving me the speed up with respect to the previous version of the code right with this we will end this lecture thank you.