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

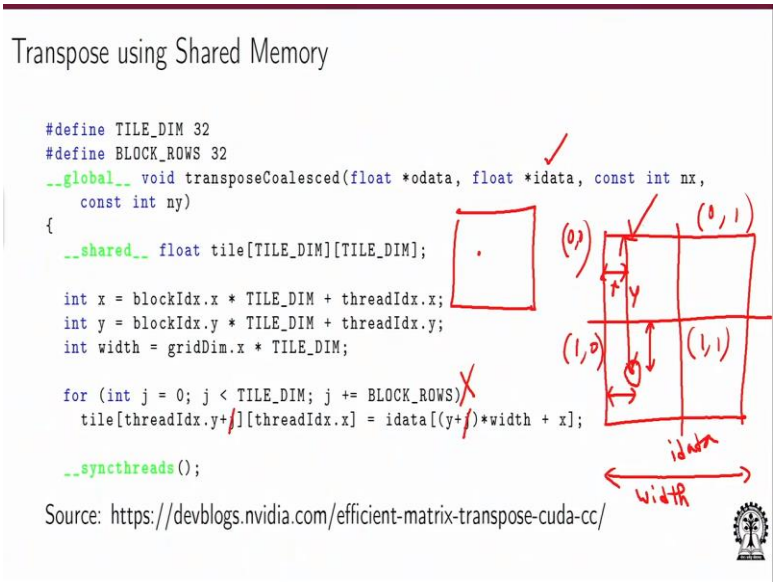
Module No # 05
Lecture No # 25
Memory Access Coalescing (Contd.)

Hi welcome back to the lectures on the GPU architectural and programming. So in the last lecture if I recall correctly we have been discussing certain different possible ways in which we can actually implement the matrix transpose operation in a GPU. So we actually provided 2 possible option. One was you load by row store by column and load by columns store by row two possible different ways to write that matrix transpose operation.

And we should saw what are the possible ways in which the computation gets affected and all that. We will continue in the same frame and we will today actually explore further into how transpose can be accelerated using several optimizations which are possible. So the first optimization would be if you remember correctly in the previous approaches we did not make use of the share memory.

So the first optimization would be that you do not directly load from the global memory and then store back into the global memory. But rather you load in terms of times into the shared memory and then you write back into the global memory.

(Refer Slide Time 01:40)



So let us get on with the first example on computing transpose operation on matrix while exploiting shared memory optimization. In the program example that we have provided in our case we have written the tile dimension and the number of block rows to be equal. So to be more specific here let me just give an example what we mean by this 2 parameters. So essentially just like our earlier example where we exploited share memory if you remember it was in matrix multiplication.

So again we will be using the similar concept of tiles and for that we will define the tile dimensions and this space we are considering 32 cross 32 size tile. So we are considering 32 cross 32 cross tiles. And now when we start talking about the number of blocks and what is the size of the blocks? In this specific example that we will be giving we are again considering the number of rows in a block to be also 32. But if you look into some other similar examples for this same problem like for example in the source that as been provided.

You will see that essentially they will consider the thread blocks to be a size 32 cross 8 now there is a reason for that we will come back to that. And there is also reason why here we considering both as 32 we will come to that. Here I just want to make clear that there is a difference in the way you will see the example done in the references in the way we have done in the example. So coming back to our idea of doing a matrix transpose while exploiting shared memory.

So the whole reason for using the share memory would be what? Like if you have seen that other naive column and naive row approaches there was either a problem with non-coalesced loads in terms of global memory coalesced or there was a problem with non-coalesced stores right. We want to elevate the problem and use the shared memory to avoid this problem of global memory coalesces either in terms of loads or in terms of stores.

So what we will do is instead of taking any of those approaches that you load of a full row of data from global memory and storing into a storing back to global memory into a full column or the reverse rather we store at a time a tile of data. So if you just try to give a pictorial representation here. So we will divide the entire memory space in terms of tiles and then we are saying that ok the first thing I am going to do is I will load a tile of data from the global memory location of the matrix to the shared memory.

So for doing that the first if we just quick on following of our example. So here the we define the tile dimension 32 cross 32 and we are considering blocks to be also of the same size as tile dimensions. That is whether the number of rows in the block is also 32. So that would mean in one load operation I have this many numbers of threads. These 32 crosses 32 number of threads which are operating and loading a full tile into a sharing memory right.

So I have got this many number threads here. Now the threads would simply get they are corresponding x and y values that each thread will work with which point in the global memory the corresponding x and y coordinates. It can just compute from this expression right. So consider for example here so this is a block with id 0 0, block with id 0 1, block with id 1 0, block with id 1 1 right.

So m I have the block id x dot x impressing this way and block id x dot y impressing this way and again inside the block I have the thread id x dot x impressing this way and thread id x dot y impressing this way is that 2d representation here and so the overall width is of course given by the grid dimension that is when I mean if I take a simple example it is going to be grid dimension the x direction is a number of block in the x direction times the tile dimension which is obviously here right.

So essentially that would be 32 cross 2 cross 32 cross 2 I mean in each the so the width essential is 64 for this simplest example I have taken here right. So now what will happen is here I am trying to run a loop using which 1 thread would be loading the values corresponding to its location. For our example first of all let us forget this loop we will come back to this and we will also see why this loop is ridden it for our purpose.

So if I forgot this loop that would mean I can also forget this j here right. Now look at the code so essentially all that is happening is so this i data is your input matrix which has been passed right. So essentially you have computed for each thread you have computed x y coordinate right. So let us pick up some place here so this is my y and this my x. So I have computed this x y coordinates and accordingly this simply this expression is simply giving me y times width + x of course the width is the grid dimension that is 2 blocks and right.

So that in the x direction times the tile dimension so this is my width right. So multiply y by width + x that gives you exactly this location. So essential you load this thread you load this position from the i data, and it would write into the corresponding tile right. So let us say this is my tile and in here where is width going to write first of all. So of course, a now if I look into the corresponding thread id for this x for this thread this corresponding location.

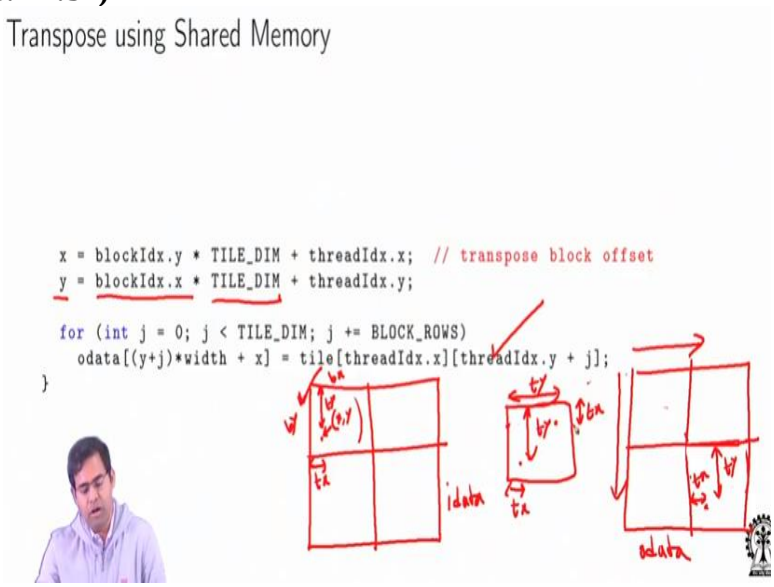
So the way I have computed the x and y values is essentially the offset of the block in the x direction multiply by tile dimension + thread id x dot x right. So this essentially would be my thread id x dot y and this essentially would be my thread id x dot x that is the offset inside block right. That is the exact location that is the corresponding matching location in the tile in the shared memory where this thread would be loading the value right.

So essentially I have this tile which has been defined and what the thread is doing is? Doing a load of this tile right. Now let us come back to what essentially is going on here in terms of the loads by the threads right. So every thread is going to execute the same code here it will compute exactly in which location in the tile is going to work with right. So overall you have this shared memory that has been defined right and then when you have this code here this thread is essentially computing where exactly it is going to load from the input data.

And then the next part which is interesting here is that the thread now has to figure out that where in the output matrix is going to write the data that it has loaded in to the tile right. I hope this part is clear to everybody. Now observe the interest thing here you have defined the shared memory right. As we know the share memory is define in a way that it is transparent to the entire block of threads right. In this simplistic setting that we have taken the block size is technically here equal to the thread dimension that is what we are considering.

So essentially you have a block of threads which are each copying something one element into the tile and that way in each thread inside the thread block is actually loading some element into the tile right. And up to this point we do not have the problem. Essentially all the threads are doing sequentially row access in the global memory. And there actually writing into the shared memory again using a sequential row axis as you can see from this expression right.

(Refer Slide Time: 11:52)



So up to this is fine. Now let us look at the next step that we do now the thread has to figure out where it is going to write in the output matrix. So again let us reproduce the previous figure let say this is my so a thread block has actually copied the entire tile here right this tile and then it is now identically copied into this shared memory which is offered as a consistent view to the entire thread block. So this is my input and this is my output matrix.

Now the way we are going to do this first thing the thread has to figure out is that ok I have taken the data from one specific block in the input matrix the output is going to be retain into which specific block right. For doing this so essentially if we understand that everything that is inside

this block. Let us in general write this as a block i, j that would be flipped right. So essentially now I am I should ideally writing it to j, ith block right.

So this has to be flipped that means technically whatever was my block id x dot y should provide me the new block id in the x direction rite. So to do this flip of the block I recompute my x and y using this formula. Let us look at it very carefully. So I am computing the new x and y values where this thread is going to write. The way it is computed is so x is now the original block ids yth position. So if we just try and understand this so this is the i, jth position let us say.

So here I have a block id x and block id y so or a if I just keep it as same with original symbols that we are using in the program let us just call it x, y so this is the original x, y now the way which we have divide this is we have a block id x and the block id y and then again inside it I have an offset thread id x and thread id y right. So the first thing I have to do is have to flip the block right.

So that would mean whatever was my original block id x should now start to become my new block id y right. So that is why when I am calculating a new y value, I do a block id x times the tile dimension right. So essentially if this was my original block id x this will now be consider to do to calculate the offset of the block in the y direction. So that is why I do a block id x times tile dimension. And similarly whatever was my original block id y would now we use to calculate the offset of the block in the x direction.

That is why the block id y is being multiplied by the tile dimension to calculate x right. But the thing is when I am calculating the x yth position, I do not change the x y coordinates original position relative to the block. Let us understand what it means. So fundamentally speaking if this was my x y position then here the actual transpose position should be what in order to find that out I should have actually consider the original x and original y and just use the representation in the opposite right.

So the x would become the y and the y would become the x right so if I just consider the previous values of x and y which have been calculated following this expressions here right. So that x is used by block id x times the dimension + thread x. So this entirely be should be combine new y. So that would give me the exactly position where I am going to place the transpose but

we are not doing that let us we will understand why we will not be doing that. We just see that they can be done in a opposite and hierarchical way.

What we are doing is the first thing we will achieve is we will just first flip the position of the block. If this is the original block position, we will find out what is the alternate block position right. So when we are figured out the alternate block position we will compute the x and y coordinates of the corresponding x y values in the original by keeping the relative position here of the x and y same. To be more specific consider this was your original tx and ty.

Now you are in the flipped block here but in the flipped block when you are calculating the tx and y position you do not change your tx and ty. You consider the same tx and ty position. So from something here you come something here so the relative position of this point with respective this block is same as the relative position of the original point which is it this one block just the block position has been transpose.

There is a reason why we are doing that. Now again look at the code that is following. Again for our purpose we are just ignoring the loop. If I ignore the loop I can ignore the j in the expression that has been given right. So now what you have in the loop. First thing is even before looking into the expression consider the scenario if I would have just switched the original x and the original y. Then what would I have done?

Inside the tile I would have just gone through the corresponding position of the threads and loaded them into the corresponding output data by using the completely flipped x and y right. Just flipping the original x and y but we are not doing it I will just reiterate. What we are doing is we are just computing the x and y in such a way that the relative position of the x and y inside the block is sim but the blocks original location has been flipped right. Now what we are really going to do.

So this is the tile which is containing the data for this block. Now this tile is going to be return here right. Now for the thread which is working in a part thread basis it has computed using this x and y relation that where is it going to write? So it is going to write here right. But the question is what is here? What is it going to write here? If it is writing the location of the original point then it is the same data right.

Because it has got the if I just go by the tx and ty values then this data is here resident in the tile and I would be writing it using the x and y values. But that is not what I will now doing the operation observe this expression very carefully. So here from the tile when I am reading from the tile I am just flipping the position of thread id x and thread id y. If you see our original code so this is our usual access expression from the input data right.

Of course you have the y that is the row number multiply by width + x. but here and of course you are writing to the y the 2 dimension tile the thread id x dot y is there. And then you have the x for the column. But now here you use the x to check the tiles except row instead of using y to go the tiles yth row. Essentially means that you are not using data from here the identical location to write to the flip block because then what would happen you just change the position of the data with respect to changing the whole blocks position but you do not do the complete transpose.

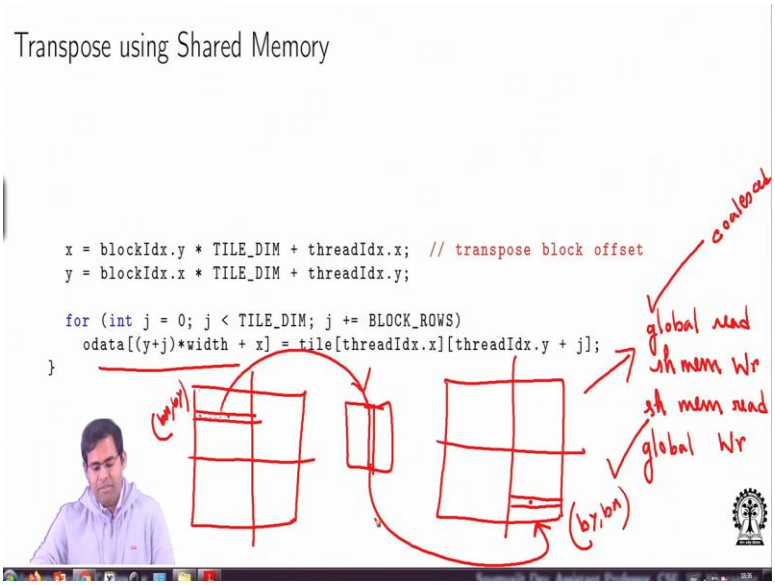
That means you do not just flip the x y coordinate into the y x coordinate. But you actually want to do that you do it in a two phase way. So what you do is? The first part is you change the flip you flip the position of the block and then you write into this position using this thread but from a different position in the column from where do you write exactly. So you when you write you load the data from the tiles txth at row and tyth at column.

So that would mean you read from some where here so then you use the earlier x offset to calculate the row offset and you use the earlier y offset to calculate the x offset right. Just see the positions are flipped. So here first thing I do is I use a different value of x and y here so essentially the thread will write the same location in the block but it has a whole flip the location of the block.

Where all the blocks location is being flipped but inside the block I am writing in the same offset location but then I do not write from the identical location in the tile but now flip the input location in the tile. By doing this double flip I achieve my overall objective of doing an overall flip from x y to y x. Question is why really, I am doing that ok? So now observe now the axis pattern that you have.

So from the input data you are writing to the tile so all the global loads that you have the global loads are now all coalesced right. Because you are writing to the global memory with a sorry reading from the global memory so let us let us observe what are the reads and writes going on in this instruction.

(Refer Slide Time: 23:54)



So here you are doing in terms of memory transactions a global read and then a shared memory right. Now the global read is coalesced because you are accessing the global memory row wise consecutive location right so this is coalesced right. And then this is some simple computation of x and y but now when I am writing to the flip block I do not just interchange x and y and make it y and x instead of that I use a modified x and y which gives me the flip block. Inside the flip block I write from the flipped location in the tile right.

Now why is this a good thing see what are the operations that I am doing here. So now I am doing a shared memory load. So shared memory read or load right. Now one may argue that ok you are loading by column from the shared memory because you are using x here and then you are using the y here right. Why is that not so bad thing well this is shared memory. So essentially you are not having coalescing but that is from the shared memory. So the load times are very small with respect to the global so I am ok with this right.

So these are not coalesced. But as a by product of this double flipping now when I write to the global memory that is also coalesced right. So I do a global write which is again coalesced right.

So now the good this of this two level transpose is that in between I have used a shared memory but I have used the shared memory very intelligent way. So if I just re write this was my input space and then I am writing to the output space.

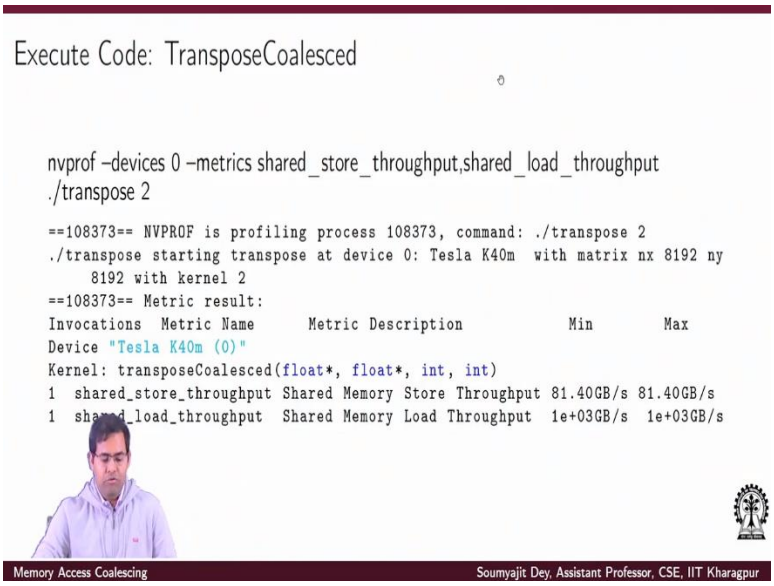
So I load from a block into the shared memory tile and then this shared memory tile which is visible to every thread I will just repeat inside the block it will write into the flip location into the block right. But again my write location in the output data matrix is again accessed in a nice row measure fashion. When I read from the global memory the warps are accessing this consecutive location in a row wise nice coalesced fashion.

This are all coalesced this are coalesced the way I am achieving is I do not write the data in such a way that I load here and then when I write I get a conflict in order to avoid the conflict I calculate the transpose position of the block. I write in the transpose position of the block. And I write in a coalesced manner and still for achieving the transpose I write from the shared memories column wise access right.

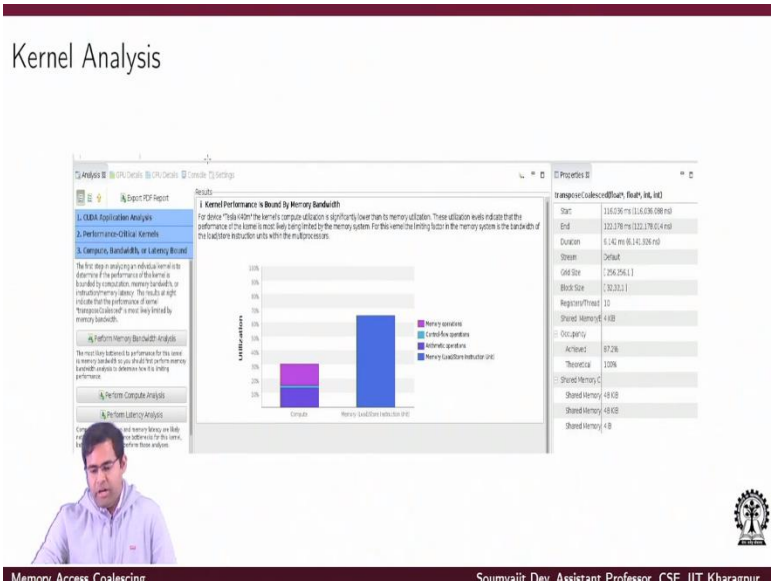
So I flip in terms of the position from where I have read right. So there is a 2 level flip here which is helping me to get the overall flip in the position. The first level of flip is with respect to the block id. The bx and by of the block gets flipped so from block bx and by I am writing into the block by, bx and when I access the positions I load data from the shared memory in a column wise fashion.

This 2 level flip helps me to achieve the overall nice coalesced behavior with respect to global memory transactions. But of course, 1 (() (28:11) you can see there the shared memory will have lot of conflicts here.

(Refer Slide Time 28:16)

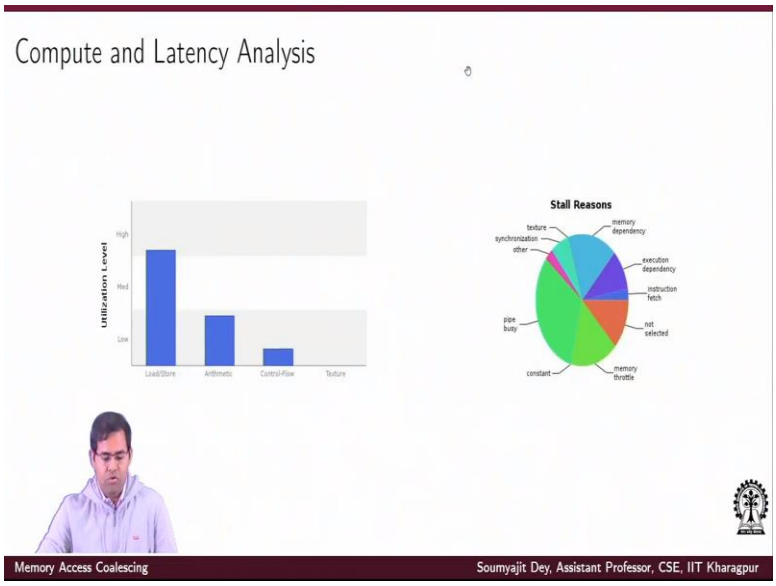


(Refer Slide Time 28:22)



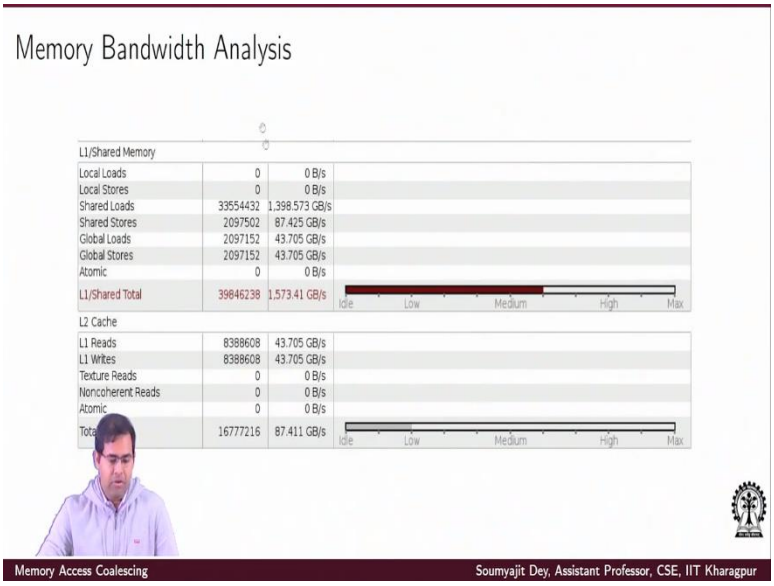
So if you just use our earlier methods for doing device able profiling then we can get the statistic we will see that this is much less if you compare with the implementation where we completely avoid using the share memory this is much less.

(Refer Slide Time 28:36)



And also if you go to the latency analysis you see the amount of memory throttling is less right. Because all your global reads and writes are coalesced, and this is one of the basic reason for that.

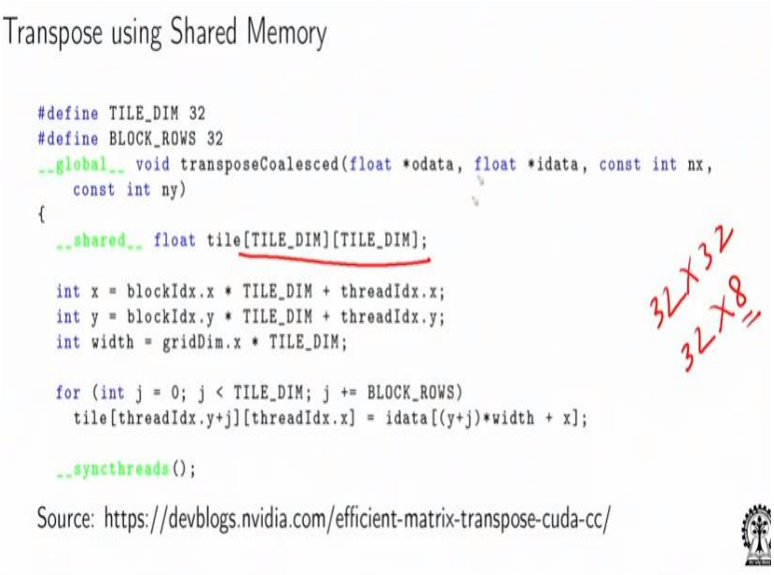
(Refer Slide Time 28:48)



And if you look it into the memory bandwidth analysis you see that the number of global reads and stores are small. However if I look into the share memory access the number of share memory stores are same as the global memory loads. Why? Look back to the code you will see that there is just like a global memory loads are coalesced. The share memory writes are also coalesced because they are writing by the row.

But when I load from the share memory as we discussed earlier, we loaded column wise. Now due to this column wise loading there has been conflict right.

(Refer Slide Time: 29:22)



Now just we will just take a look back into the origin code that why effectively we did not have a loop ok. The reason is in general when you are writing the code here one problem the way we discuss is we bypass the loop. If I bypass the loop then I have very small path thread activity. Now I if the path thread activity should not be too small so that I mean if I am considered very large matrixes right.

So increase the path thread activity one standard way that people follow in the examples is that ok you define data that share memory location in terms of 32 cross 32. But you define blocks off size 32 cross 8. So essentially you have each thread working on 4 consecutive locations or 4 specific locations in a tile of data. So then your data space size remains the bigger 32 cross 32 but you use 4 less number of threads for doing the operation right.

Now if you are doing that then the loops will actually be useful then as you can see what will happen is inside this for loop you are iterated j starting from 0 in every iteration you are increasing by a block row number assume that the number is 8 then your loop really iterates. So in so you have each thread loading 4 data points in to the tile. And then in other loop it is writing 4 data points into the global memory.

Of course, for larger matrixes you can actually it gives the different implementation you can have different possible factors here. Just for our simple example first we wanted to bypass the loop. So we consider tile size and block size similar it as I am saying that you can have a smaller size block to increase the part thread activity. So if you have a smaller size block then you have a this tile dimension here.

You have the j iterated increasing by 8 the loop iterates for 4 times and each thread load 4 elements into the tile and similarly the threads here we write 4 elements into the tile. So with this we will end this lecture where we actually a show a usefulness of share memory how it will helps to transpose to coalesced both the load and stores while doing a matrix transpose computation. In the next lecture we will look we will take a deeper look into this thank you.