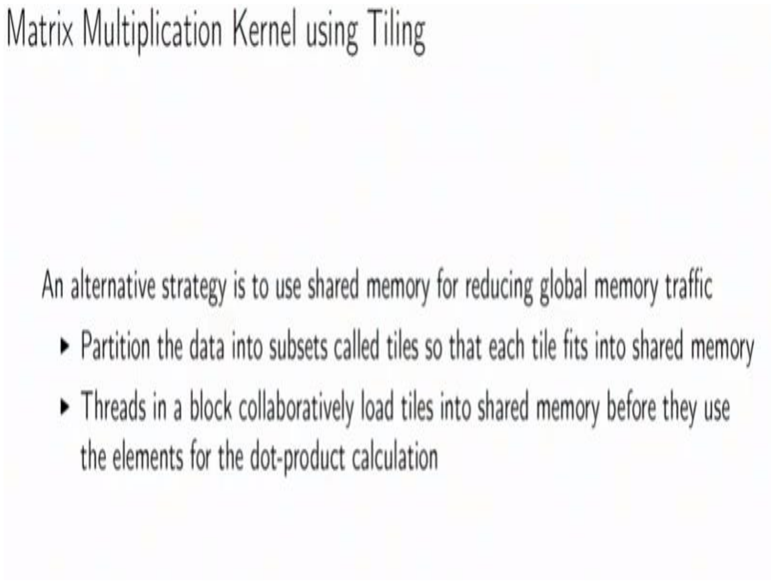
GPU Architecture and Programming

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

Module No # 05
Lecture No # 22
Memory Access Coalescing (Contd.)

Hi welcome to the course on GPU architectures and programming in the previous lecture we have been discussing about how matrix are simple idea of matrix multiplication in using a GPU can help in optimizing I mean how that can be accelerated by using some specific (()) (00:44) like the shared memory based optimization and also using the idea of global memory coalescing.

(Refer Slide Time: 00:5)



Matrix Multiplication Kernel using Tiling

An alternative strategy is to use shared memory for reducing global memory traffic

- Partition the data into subsets called tiles so that each tile fits into shared memory
- Threads in a block collaboratively load tiles into shared memory before they use the elements for the dot-product calculation

(Refer Slide Time: 01:04)

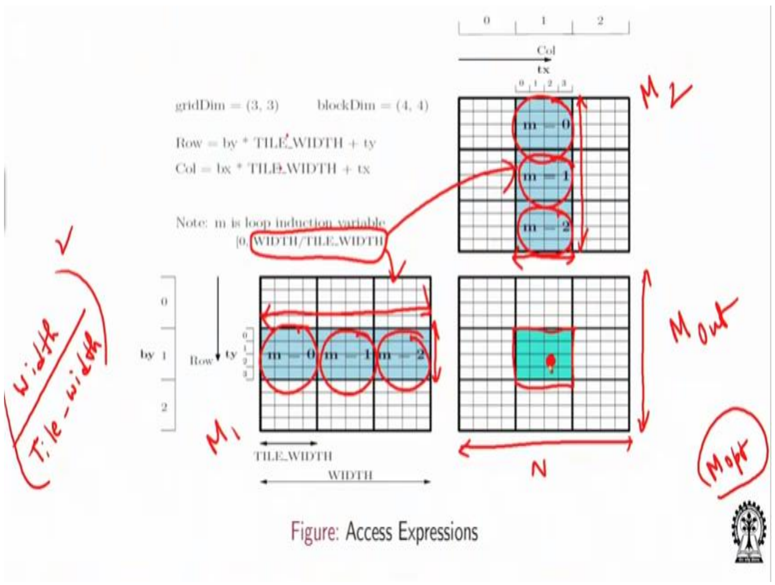


Figure: Access Expressions

So based on that let us look into this example now first thing we will do is we will define a notion of tile essentially a tile will be what same as a block size. So we will be considering 2 dimensional blocks so that is why we will define the grid of the that is the number of threads I am going to launch is still n square so that is the dimension of the matrix but the launch parameters I will be doing of very careful selection.

So what I will do is I will define as we have discussed something called as tile which is essentially equal to the block size. So in 2 dimension I will decide the size of the tile based on the size of the shared memory we will soon see that how can be done. For now let us consider that the tile size is the block size and this is an example picture we are showing so the grid id defined as a collection of the 3 cross 3 that is 9 number of blocks distributed in 2 dimensions and inside each block I have a 2 dimensional arrangement of threads in 4 cross 4.

This is an example where essentially I am holding N as 12 this is of course the simplistic example. As you see that it maps nicely with the original idea that we are considering 2 dimension matrixes so will definitely consider the blocks in 2D and also the grid I mean also inside the block inside the block thing are in 2D the blocks are also packed in 2D because that would give me very nice access expression.

Now how do I figure out some specific row and some specific column for the given thread and again that would be quite easy. So essentially here we are setting the block dimension in the x and y access both equal to the tile width variable now of course for a it will depend on the dimension of the matrix here. So the tile width variable is going to give me the block dimension in the y direction and similarly in the x direction.

So if you replace this block dim dot y and this one will block dim dot x that gives me the usual access expressions for the row and column for a specific location in the matrix right. Now how do I figure out that in which tile I am located so I want a loop to decide that which tile to load or which tiles to access right. So essentially the basic principle between inside this computation is as follows that threads computing inside a tile would do collaborative loading and computation.

So earlier in our naïve algorithm let us say any 1 thread the ij thread was bringing data form the ith row and the jth column multiplying and accumulating them to compute the ijth location. But now we are saying that no we will do something different instead of this entire tile or the corresponding whatever the threads are there they would collaborative access collaborative computation.

And when all the threads computing for this to finish they have done with computing all the elements for this tile in the output matrix. Now for the doing that what is the region of access required in the 2 input matrixes so of course as you can see that for a single row it would have been a for a single location it would have been a single row and single column but a region it would actually spread for all the locations inside this shredded output tile I would need access in M1 in all this region it is entire region right.

And similarly here I would need access in this entire region right for the issue is not still fixed how I am going to do some optimization and decrease the number of loads there are going to happen in the global memory that is the key thing they are going to do right. Just another point we like to make here let us suppose somehow we are figured out how to do some optimize computation and things inside this output tile.

But then let us call it some method 1 some method optimize method right so if I am using this method how many times I am going to call this optimize method how do you figure out that earlier I was computing for each location and that I was doing M square number of times but now of course for calling if I have think I have to discussing that if I am using this optimize method for doing computation for one tile here number of times I am going to call this method would be the total number of tiles here right.

So what is the total number of tiles here width by tile width square right because as we can see in this example that since my tile size we will be divided by the width I mean considering width and length both same here. So I have got to access that many tiles here on the x side and the y side you multiply them so you have to access this many tile right. Now if I think that okay for so this M of pt would be accessed this many times.

Now let us try and look into the functioning of this optimized method right so first of all since I have broken down the matrix into this kind of coarse drain distribution of blocks which are now calling them as style. No observe that for computing this output tile I need a loop which should iterate through M1 and access 3 types because this is the region where mark which needs to be accessed this region can be nicely broken down into 2 types.

As per this picture in general I can say that I can be nicely broken down into this width by tile width by number of tiles right. So for computing this output tile I need to access this width by tile width the number of tiles from input matrix 1 and similarly I need to access this width by tile width number of tiles from the input matrix 2 right. However have to figure out what is the location of each this tiles so suitable variable in this as I have to load this tiles into the memory and then I have to do some computation right.

So I can say that all the threads which are computing for this block the output tile the first thing they have to do is they have to load let us say I said the loop iterated variable name as 0 and I load the tile 1 here and the tile 1 here and then I do some computation using them then I load the next tile from here and the next tile from here. I do some computation using them then I load the next tile from here and the next tile from here and again i do some computation using them.

And finally all this threads inside this block they are ready with the output so I am loading in tile sizes and I am doing some computation using the data then again I am loading in tile sizes I am doing some computation using the data and that is how I am going on. Assume that for the time being assume that this is going to work right.

(Refer Slide Time: 10:14)

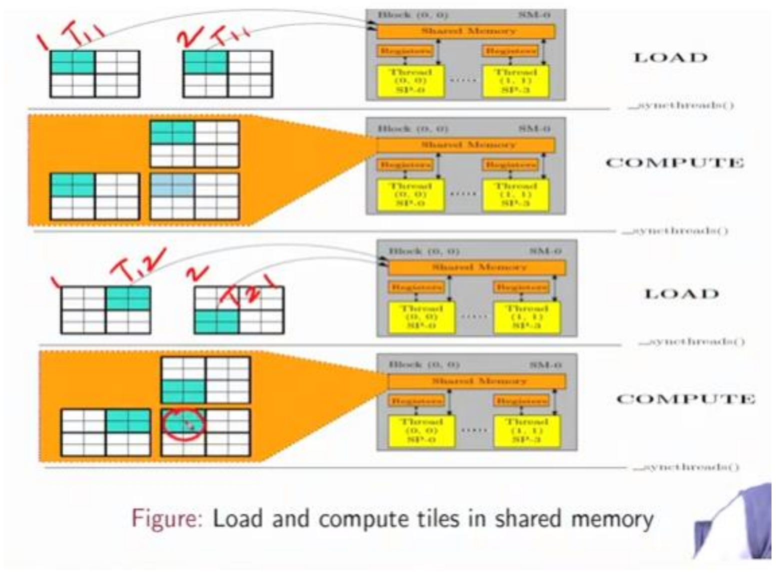


Figure: Load and compute tiles in shared memory

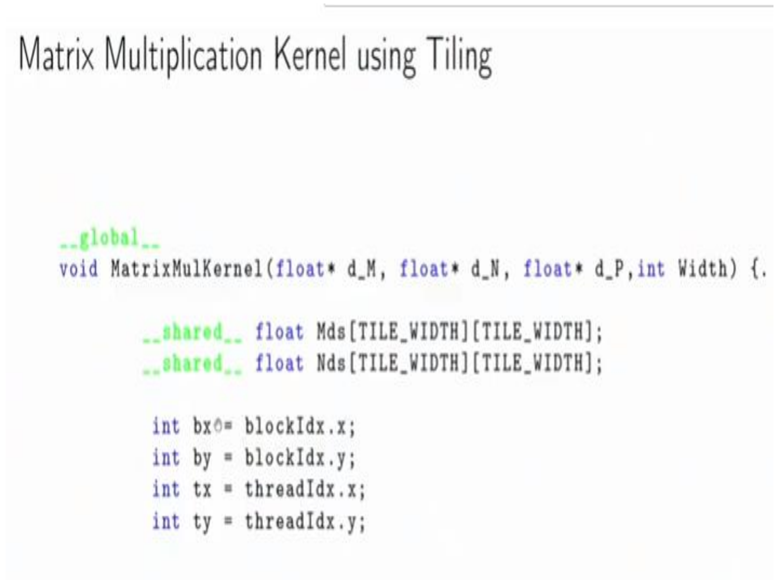
So if this idea is really going to work then I would essentially have a sequence of load followed by compute again load followed by compute right. So essentially I am using a set of threads in the same blocks to load to tiles right so I have effectively loaded this tile and this tile and did some computation with them and in that way I am making this progress right. So I have loaded this 2 tiles and made some computation.

Then again I have loaded the next tile this one and this one and they have highlighted here and next tile this one and this one and again I have done some computation so this picture is show using smaller size. Let us say I am showing a instead of having this kind of 3 cross s number of total tiles in this picture for representation purpose I have 2 cross 3 number of total tiles so I am trying to show that suppose that measure works then essentially the threads which are doing some computation for this region they are going to collaboratively load data from this tile of matrix 1 and this tile of matrix 1 to the computation.

So this is the load space and using the loaded data in the shared memory they will do some computation then again is the next load phase. So this is matrix 1 this is matrix 2 so they are loading the 2 tiles tile 11 they have put them into the shared memory and did some computation then they are loading tile 12 and they are loading tile 21 then again they are doing some computation and with all that the final result is ready.

But the question is what is the good thing here how does this really help still we are assuming that this sequence of load computes are going to work we will see that why it works.

(Refer Slide Time: 12:26)



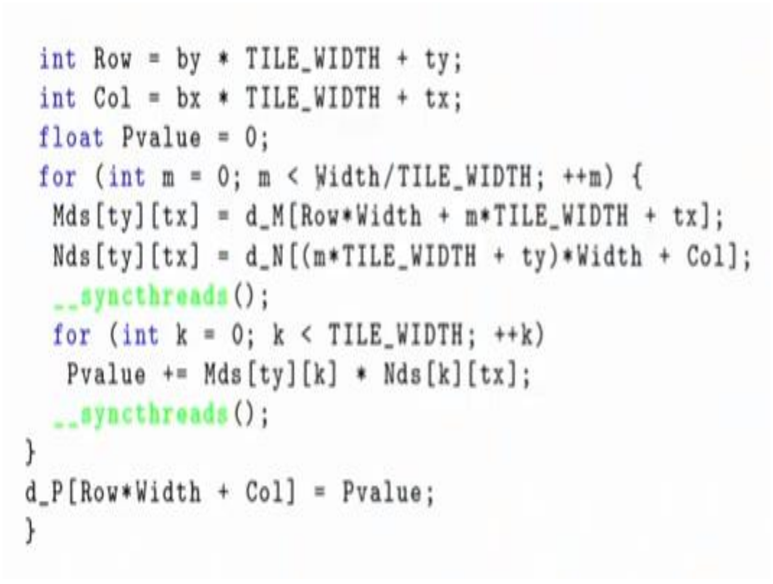
Matrix Multiplication Kernel using Tiling

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P,int Width) {

        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
        __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

         int bx = blockIdx.x;
         int by = blockIdx.y;
         int tx = threadIdx.x;
         int ty = threadIdx.y;
```

So suppose that works then would write the kernel like this so inside the kernel I would define 2 shared memory locations each which are of the tile size width of the tile width why? Because assuming that I am going to do this kind of load compute I am loading this 2 types from this 2 different matrixes input matrixes into the shared memory right the compute phase will go in the shared memory and we will see the advantage out of it right.

So the 2 tiles have to be loaded by the set of thread who are working for this part of the computation so the set of threads are going to first load this tile from matrix 1 and this tile from matrix 2 into the shared memory so I define to share memory location each of the size that should be able to hold 2 of this tiles together right. Then I get the block id is here thread id is here in this smaller space variables.

(Refer Slide Time: 13:31)



```
int Row = by * TILE_WIDTH + ty;
int Col = bx * TILE_WIDTH + tx;
float Pvalue = 0;
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
 Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
 __syncthreads();
 for (int k = 0; k < TILE_WIDTH; ++k)
  Pvalue += Mds[ty][k] * Nds[k][tx];
 __syncthreads();
}
d_P[Row*Width + Col] = Pvalue;
}
```

And then I have to perform the actual loads right. So to perform the actual loads I first figure out the row and the column index for the locations. So essentially what does this mean I am essentially putting in here to variables row and column which will actually so if you just take the example of what is that what do I really get out of this block id y and thread id y of any thread.

(Refer Slide Time: 14:09)

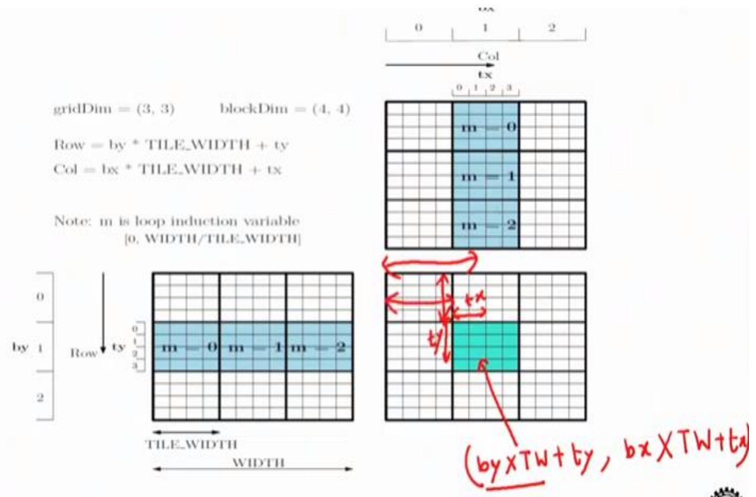


Figure: Access Expressions

So any of the threads are trying to figure out that okay what is the location for this it is going to load right so block id y times the tile width plus the thread id y comma this is block id with respect to x. So what is this block id times the tile width so you come up to here and then the thread id x right. Why I am really multiplying the block id with the tile width basically that gives me to figure out exactly I mean okay this is the so if I multiply this by / tile width I get to here.

So I get to this much and then I shift by the tile width similarly with bx times tile width I get to this much and then I shift with the tx right. So this tells me to figure out for each thread which locations of the input memory is going to access right so as we have discussed that all the threads who are working on this output tiles they are going to bring data column comparatively from 2 of the input tiles let us say this one and this one or if you look at the simplistic picture from this tile and this tile for first of all you would figure which of the locations of the threads of the input tiles they are going to access.

So for that they simply use the row column indices and they using this row column indices they would compute the corresponding location in the original input matrix. So essentially as we can see the row will get multiplied by the entire width of the matrix so that gives me row the position in the row and then with the loop iterated variable the tile width and then I get into the tile with the tx position.

So I hope this gets clear now so first point is since I am going to do this inside a loop in the first phase I am going to access since from this 2 locations. So for this location essentially what are the position that this thread are going to access. So in the first phase of the load compute it is going to access data from here and here. So when this loop runs for its first iteration M is 0 so row times width plus tx.

So you have already figured the row times width that gives you that position here that which row and then you shift by the tx amount that is the amount of shift you have here right. I hope this is clear because this is your tx so you just shift by tx inside the first tile for the corresponding row. So in that way the thread as figured out which data to bring from matrix 1 and similarly using the other expression it will figure out which data to bring from matrix 2.

So as we have already pointed out those we did the exact data points that the thread will bring in and finally when all the threads corresponding to the tile have executed this code what do I really have I have this situation that for all the threads in this style they have brought in this 2 data points in the shared memory from the original global memory of the matrix to the shared memory right.

Now in this tiles I have got this two data points right for the threads working here for the threads working from others tile they will be bringing doing collaborative load from some other locations right. So this threads have done this or this part of its work right they have first figured out they belong to which tile and then they have brought in data collaboratively in parallel from here and here and then they are going to get into this loop through which we will do a partial sum computation.

(Refer Slide Time: 19:29)

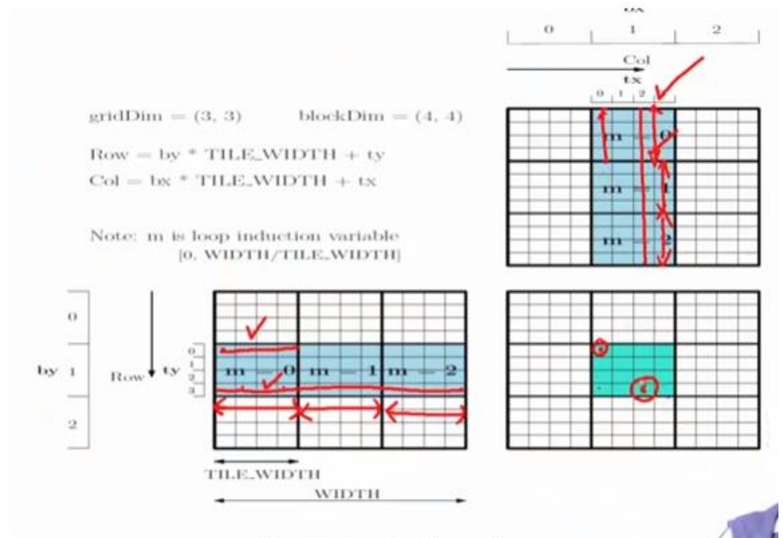


Figure: Access Expressions

So what does this mean so if you look into this picture finally what is going to be the output for this location let us take any random location the one which we are talking about. So that would require me to do an addition a multiply at for things here and here what am I really doing is I am facing this multiply at into 3 parts. In the first part I load things from here and things from here and do the multiply at in the second part of the computation I will load things from here and things from here and then do the multiply at and the last part of the computation we will do this and this and multiply at and get together everything done.

So when the threads which are working collaboratively here to bring in this tile and this tile so essentially after bringing this 2 tiles in shared memory what they will be doing here is they will be doing their part of the multiply at operations for only this parts that means for this thread this was responsible for bringing data from this location and this location. But parallelly the other threads for the other locations have bought in the data points and store them in the shared memory which (()) (20:58) this threads requires right.

So this thread will now actually use data point being brought in by other threads and to the multiply operations this thread is suppose to do but upto this much brought the entire thing together why. Because I have not still brought in the data for the other types into the shared memory right this is the main reason. So similarly what is happening while this thread is this part of computing for this location all the threads in the tile are doing their part of the computation for various other locations.

For example if I consider the thread corresponding to this point it had brought in this data and this data and after bringing in those data points it is traversing this much and this much and doing a multiply here right.

(Refer Slide Time: 22:00)

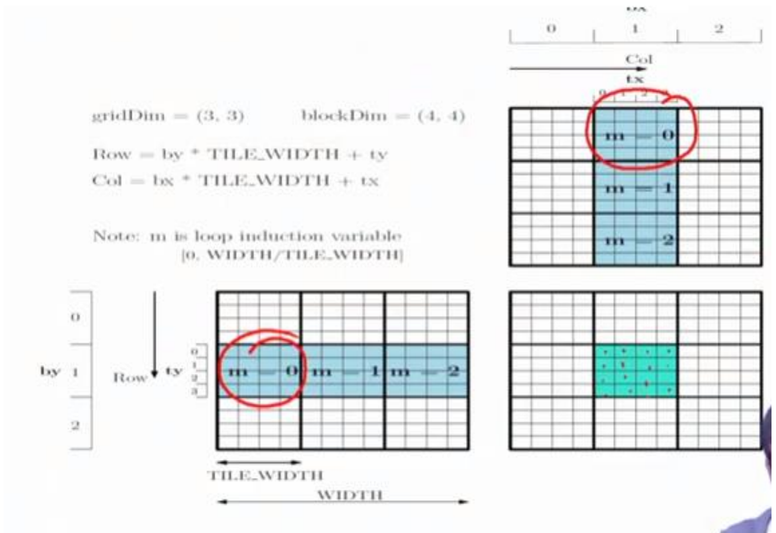


Figure: Access Expressions

So after all the threads for this tile have brought in this tile and this tile and then done the computation and what do I have? I have some data points computed for all the location but they are not the final results they are all partially computed multiply at for corresponding rows and columns. But what was the good thing here? The threads were actually using data brought in by other threads in parallel right.

So all the threads in this tile have bought in data have stored them in shared memory and now while doing the multiplied operations they have accessing data from the shared memory and they are not only accessing data that they have brought in they are also accessing data that other threads are collaboratively bought in. So with this if I progress with the code so first of all after ensuring that all the threads have brought in the respective tiles in the load phase then only I should go into the compute phase and do the partial SM computation that is why after bringing the data there is sync thread and then as long as I am not done with the partial sum computation I should not go into the outer loop and bringing more tiles right.

So that is why until and unless the partial sum computations are already done and whatever data as was brought in in the previous load phase all the threads need to synchronize only after this point what is the guarantee I have? The guarantee that I have this is right now compute having partial sums computed considering 2 of the input tiles so once all those threads synchronize there they will again revert back to the load phase after reverting back to the load phase.

(Refer Slide Time: 23:59)

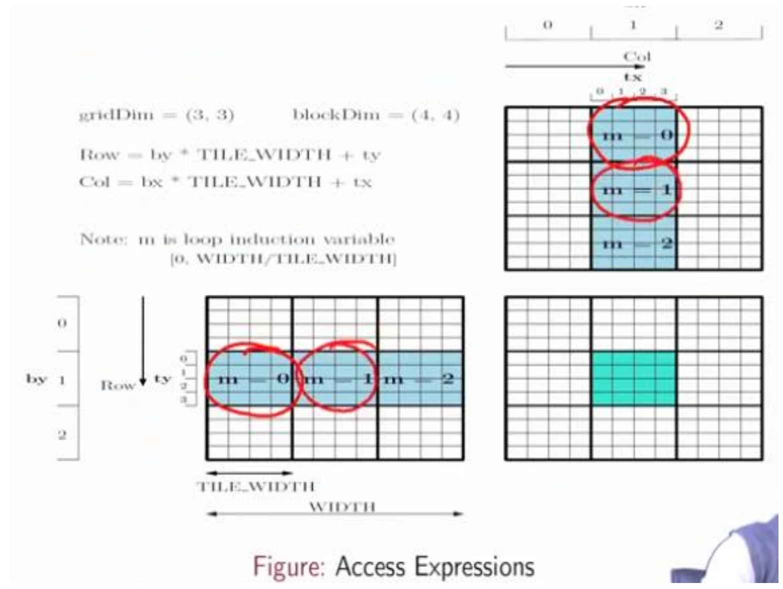


Figure: Access Expressions

So assuming that this have already been taken care of this input tiles have already been taken care of this threads would again bring in more tiles to the load phase and again start doing the computation. So that is what is happening here right so again they will go in here bring in data in the shared memory again have a synch thread when all the data points are ready and then they will again do parallelly they again parallelly collaborate into doing the computation from the multiplier.

So for this simple example is getting done right here right so since I have only 2 cross 2 number of tiles. So essentially the threads which are responsible for doing computation for this 1, 1 positions tile it will have 2 load compute phases it will load tiles from here and here into the shared memory then collaboratively compute the sum again in the facet we will load tiles collaboratively compute the sum right.

In general it will be having much more number of load and compute phases like that so overall you can see that only when the entire sequence of loads and computes are done the partial sums will finally get a update and they will replace them the final results right. So once each thread as completed its sequence of loads and computes the corresponding its own private copy. So this is he private variable right for each thread to have the final value corresponding to its own ijth location right.

So then it would actually load this final value in a corresponding location in the target output matrix in the device manual right. So this dp is the target output matrix in the device memory it is getting updated only at the last.

(Refer Slide Time: 26:02)

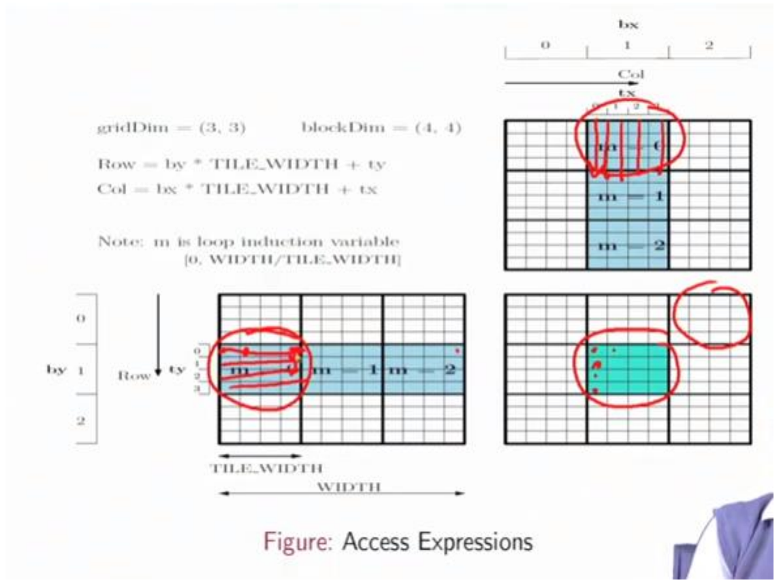

gridDim = (3, 3)
blockDim = (4, 4)
Row = by * TILE_WIDTH + ty
Col = bx * TILE_WIDTH + tx
Note: m is loop induction variable [0, WIDTH/TILE_WIDTH]
bx
0
1
2
Col
tx
m = 0
m = 1
m = 2
by
Row
ty
TILE_WIDTH
WIDTH

Figure: Access Expressions

So to make things much more clear what is happening is for doing the computation of this tile threads for this locations are making progressing parallel. First thing they did is all the threads together brought in stuff from this 2 tiles and then they made progress. For example this thread made progress like this and this in parallel some other thread for making progress like this and again in the same column like that.

But after this phase is done all the threads have covered all this positions right see everything is still going on in parallel like our original optimized CUDA kernel for parallel matrix multiplication right. Like while this thread is being computing for this tile the other thread are of course computing for their respective tiles right. But the good thing is happening is each thread is not performing all the loads it is required to perform from the global memory that means each thread is not bringing the same row of data in multiple times.

This thread as brought in this data whereas next thread has brought in next position of the data but for doing the computation this thread is using all those data. Loading them when it is requiring them from the shared memory not from the global memory and that means many number of global memory loads are getting changed to shared memory loads.

(Refer Slide Time: 27:46)

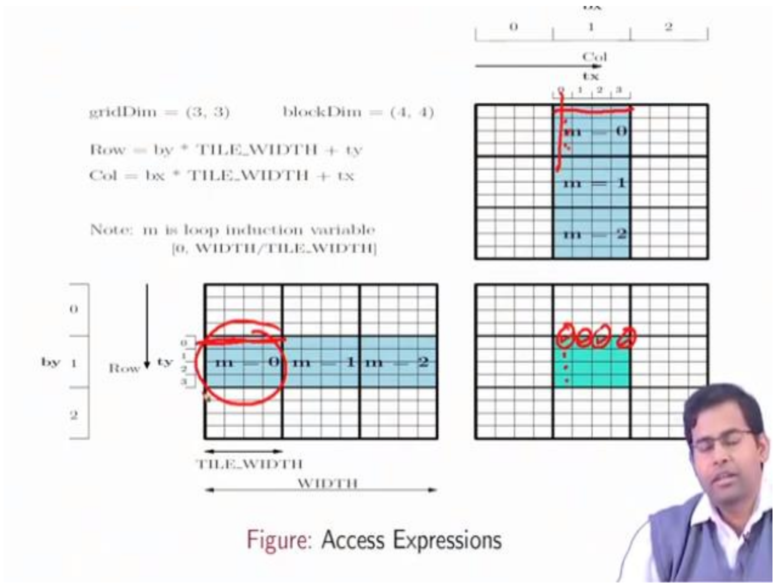
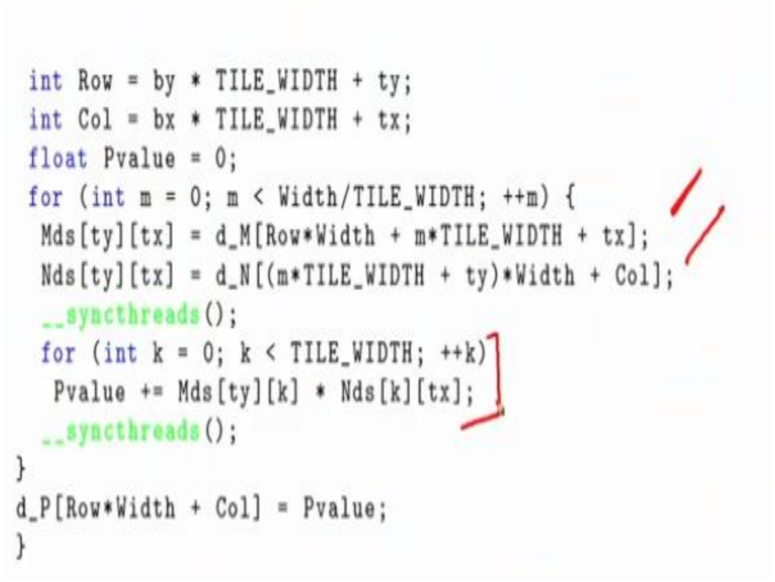


Figure: Access Expressions

As an example I would just again revisit this so consider the threads are working here so essentially if I consider all the data that is brought in by this 4 threads it would be this whereas the other threads are bringing in data for the other locations but when this threads are into the compute phase. For example the first thread is accessing everything from here and here where all this other entries has been brought in by thread of this locations.

But when this thread is doing the compute in loading data from this other locations it is doing all this loads from shared memory. It is not doing this loads from the global memory that is the most important point essentially once all the actually global loads have been performed while bringing the things for this entire tile collaboratively by all threads in parallel after that in the next sequence of multiply and add operation inside the inner loop.

(Refer Slide Time: 29:09)



```
 int Row = by * TILE_WIDTH + ty;
 int Col = bx * TILE_WIDTH + tx;
 float Pvalue = 0;
 for (int m = 0; m < Width/TILE_WIDTH; ++m) {
  Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
  Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
  __syncthreads();
  for (int k = 0; k < TILE_WIDTH; ++k)
   Pvalue += Mds[ty][k] * Nds[k][tx];
  __syncthreads();
}
d_P[Row*Width + Col] = Pvalue;
}
```

All the operations are now shared memory operations so I can now say that now I have a clear difference here all this loads these are my global memory loads. But after that all the threads are accessing data from the shared memory so all this access are going to be shared memory operations and those global memory loads what is the good thing about them so here I come to the next part the next fascinating part.

(Refer Slide Time: 29:56)

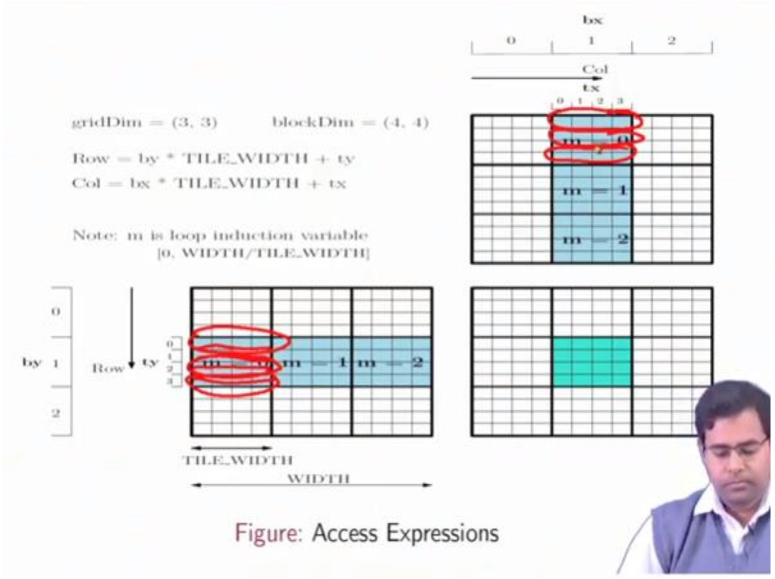
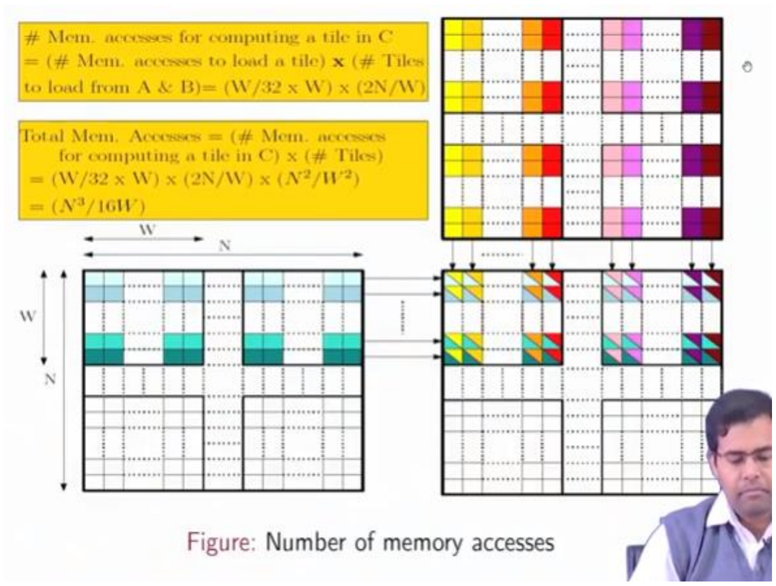


Figure: Access Expressions

So observed the load pattern so when all the threads are loading data here so I am loading data from all this places right. So all this loads will be coalesce again all the other loads will be coalesce and like that right similarly here. Since I have broken down the computation to this phases of loads followed by compute again loads followed by compute so this loads are now nicely coalesced.

(Refer Slide Time: 30:39)



Mem. accesses for computing a tile in C = (# Mem. accesses to load a tile) **x** (# Tiles to load from A & B)= (W/32 x W) x (2N/W)

Total Mem. Accesses = (# Mem. accesses for computing a tile in C) x (# Tiles) = (W/32 x W) x (2N/W) x (N^2/W^2) = $(N^3/16W)$

Figure: Number of memory accesses

So over all if I am trying to do a computation of how many memory accesses are done for computing a tile so it is equal to the number of memory accesses is required to load a tile times the number of tiles to load from the 2 input matrixes let us call them A and B. So what is the number of tiles to load? So if you think so this is for computing a single tile right.

(Refer Slide Time: 31:24)

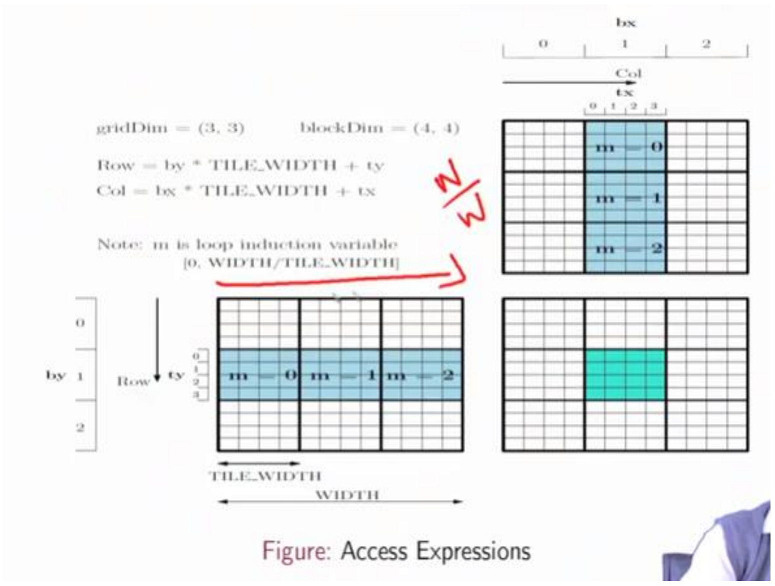


Figure: Access Expressions

So the number of tiles to load would be so this is N you divided it by the width so that gives your N / W number of tiles here and similarly here the total you have got 2N / W number of tiles to load for computing in a single tile. So you have got in total this many tiles to load and what are the number of memory access required to load a tile. Well as we have discussed that when I am look into the tile all this accesses are going to be coalesced row wise.

So the number of accesses would be W / 32 times W the times W would come for the each of the columns but each of the row I have got coalescing W / 32. So this is the total number of accesses for computing in single tile the total number of memory accesses in general would be the total number of memory access would computing a single times the number of tiles. So that would give me W / 32 times W times 2N / W times the number of tiles is of course N / W whole square or N square / W square so that gives me N cube / 16 W.

So that would mean if I can choose a W which is significantly big I can have an order of magnitude deduction here / N. So W is comfortable to N I it is a big fraction of N I can have lots of acceleration here right. So that is the primary take away of the message here that with significant amount of tiling depending on of course amount of shared memory you have available I can reduce the total number of global memory accesses for computing this matrix multiplication for a significant factor by getting the threads to collaborate work from the shared memory and also the tiling is helping me to increase the amount of coalescing that is possible here.

So that way it leads to significant reduction in the total computation time and as we know that matrix multiplication being a fundamental operation being many part of many large many significant optimization techniques and you mean work loads and many other pluses if you can accelerate matrix multiplication that really helps in so many multiplication domains. So this is one of the most fundamental computations where GPU’s really help and we will take several such examples of other computational in the next lecture thank you from now.