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

Module No # 05
Lecture No # 21
Memory Access Coalescing (Contd.)

(Refer Slide Time: 00:34)

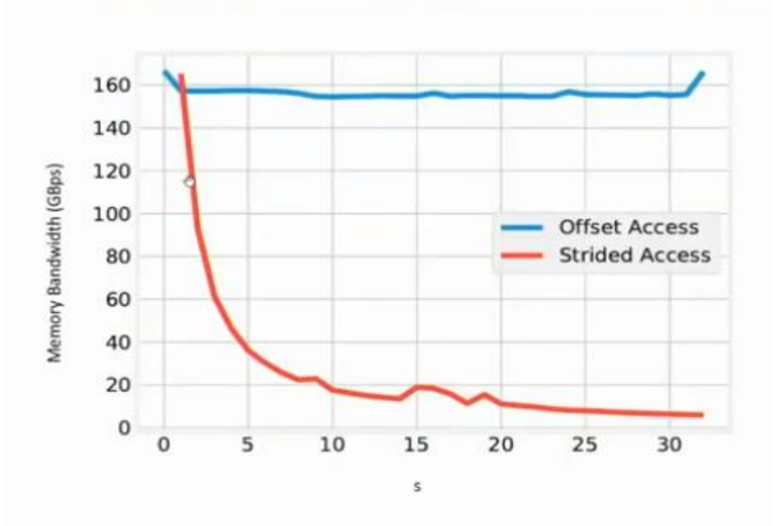
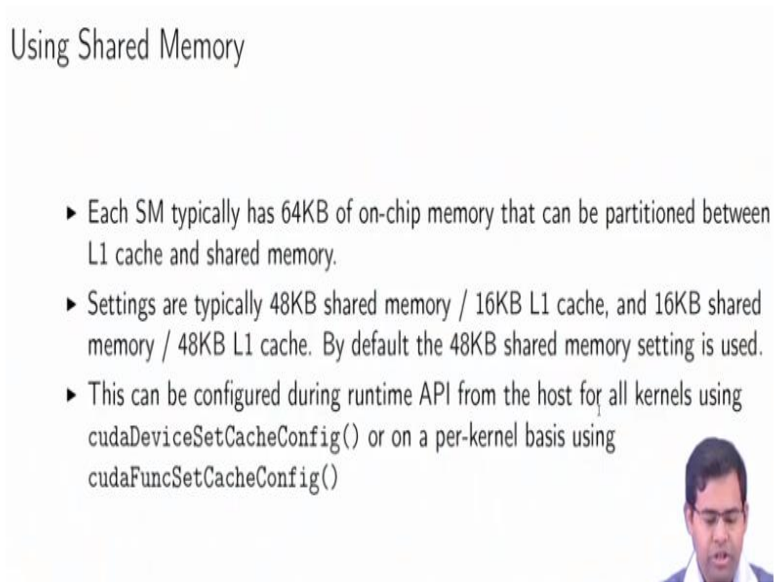


Figure: Memory Bandwidth Plot

Hi so welcome to the classes of GPU architecture's and programming in the last lecture we have been discussing about a bit on this usage of shared memory I mean how the shared memory is configured and in general how global memory transactions are really happening. If you remember the curve we have been showing that based on the different ways I am doing access of an array whether it is offset bed access or strided access we could see that the memory bandwidth that was exploited that is the number of global memory transactions per second that was having.

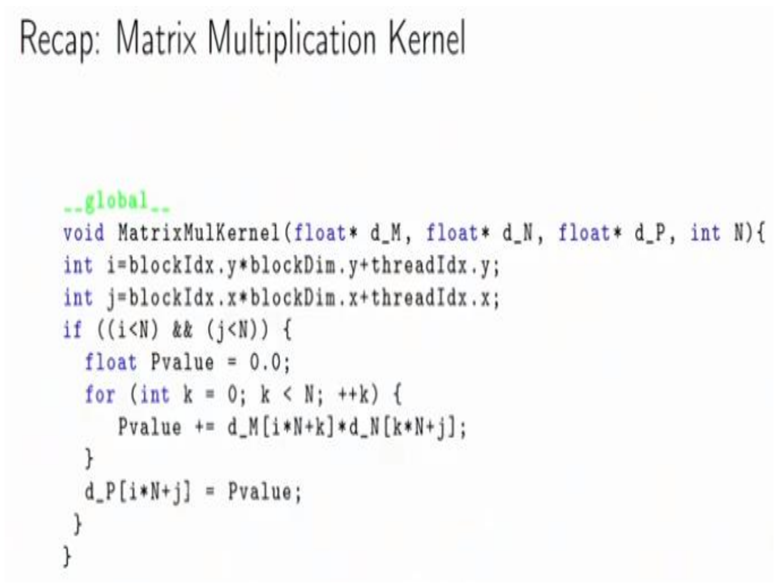
(Refer Slide Time: 01:08)


Using Shared Memory

- Each SM typically has 64KB of on-chip memory that can be partitioned between L1 cache and shared memory.
- Settings are typically 48KB shared memory / 16KB L1 cache, and 16KB shared memory / 48KB L1 cache. By default the 48KB shared memory setting is used.
- This can be configured during runtime API from the host for all kernels using `cudaDeviceSetCacheConfig()` or on a per-kernel basis using `cudaFuncSetCacheConfig()`

I mean that how it was really getting affected so after that we also pointed out that there is earlier I mean how to actually configure shared memory and the L1 cache what are the related CUDA comments for doing that.

(Refer Slide Time: 01:20)


Recap: Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int N){
int i=blockIdx.y*blockDim.y+threadIdx.y;
int j=blockIdx.x*blockDim.x+threadIdx.x;
if ((i<N) && (j<N)) {
  float Pvalue = 0.0;
  for (int k = 0; k < N; ++k) {
     Pvalue += d_M[i*N+k]*d_N[k*N+j];
  }
  d_P[i*N+j] = Pvalue;
 }
}
```

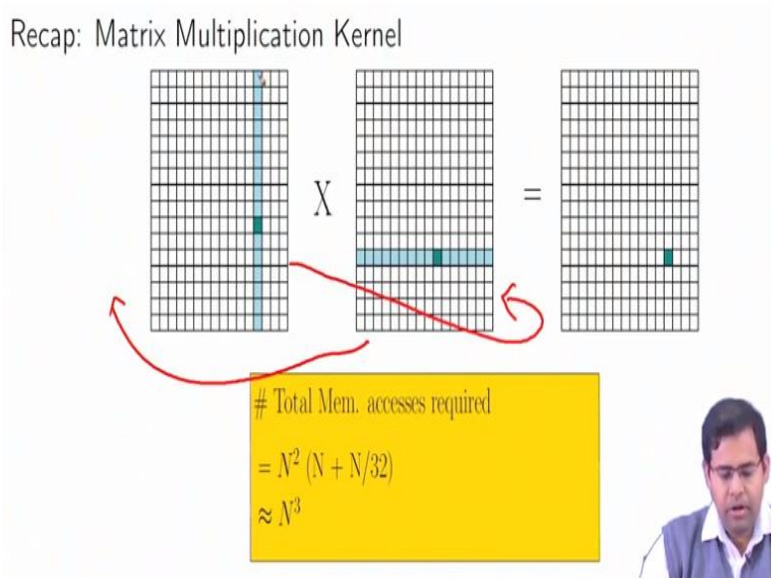
So now we will see how I can exploit this shared memory that is present in each of this symmetric multiprocessors since streaming multi processors and in each of the streaming multi processors and how they can actually help in accelerating parallel algorithms by suitable programming techniques. So just as a recap let us consider our very I mean common popular matrix multiplication kernel.

So this is the usual code of matrix multiplication kernel which you might be recall from our earlier examples so all that we are doing is there is a if statement which is basically checking whether the I and J that is the that is the location of the row and column that we are trying to access in the matrix whether they are inside the relevant bombs. By the way the row and column positions were figuring out again by using the block id x and thread id x block id y and thread id y variable values.

So each thread for its unique combinations of block id's and thread id's would able to find out a unique i and j. Then you should check whether the i and j are inside the memory range of the matrix and then it will each thread will get inside this for loop. So as you can see the for loop running from 0 to N – 1 so essentially we are talking about multiplying 2 N cross N dimensional matrixes.

So the thread would essentially multiply the elements of 1 row of matrix 1 with the elements of the corresponding jth column of matrix 2 and it will essentially sum it up store it in P value and this value would finally be transferred to target matrix location the output matrix location whose dp here right.

(Refer Slide Time: 03:24)



Recap: Matrix Multiplication Kernel

Total Mem. accesses required

$= N^2 (N + N/32)$

$\approx N^3$

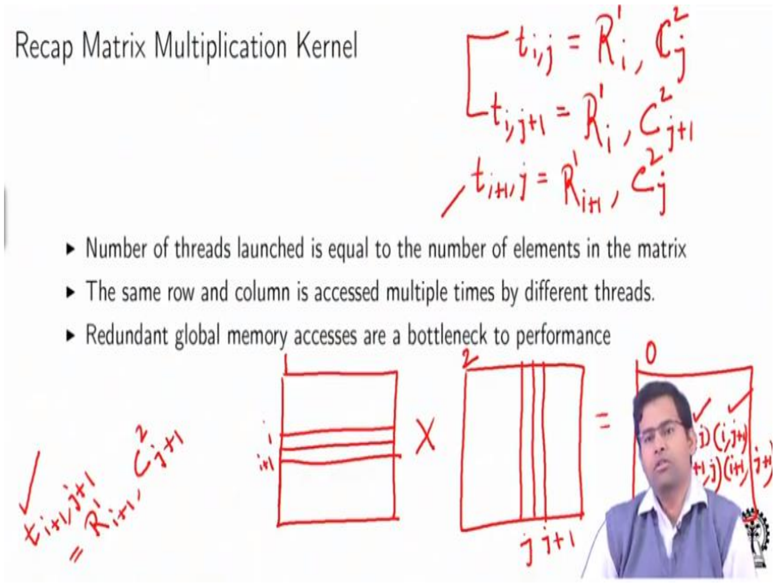
So in terms of a figure this is the part thread activity so the thread I have launched for accessing ith row and jth column so I would actually preferred this picture in the flipped way basically. So if you look into the access pattern here so all that we are doing is we are figuring out an i and j.

So i is the row index and j is the column index so just to match with it just consider the figure in a flipped way.

So just think that this is matrix 1 and the next 1 is matrix 2 so essentially the thread will navigate through row the ith row and the jth column pair wise multiply the elements and act and final figure out the final output value for the ith row and the jth column of the output matrix right. I mean this is usual way we do matrix multiplication. So what is the total number of threads that are getting launched for computing the output.

So definitely the total number of threads have to be used to equal to the number of elements in the matrix because each threads is responsible for computing 1 element in the output matrix right.

(Refer Slide Time: 04:56)



Recap Matrix Multiplication Kernel

- Number of threads launched is equal to the number of elements in the matrix
- The same row and column is accessed multiple times by different threads.
- Redundant global memory accesses are a bottleneck to performance

$t_{i,j} = R^1_i, C^2_j$
$t_{i,j+1} = R^1_i, C^2_{j+1}$
$t_{i+1,j} = R^1_{i+1}, C^2_j$
$t_{i+1,j+1} = R^1_{i+1}, C^2_{j+1}$
1
2
0
i
i+1
j
j+1
(i,j) (i,j+1)
(i+1,j) (i+1,j+1)

But observe with something important here with respect to the memory accesses perform by the threads. So the same row and column is accessed multiple time by different threads why would we say that. So if we draw up the simple figure here so for computing element i, j I would made to access the ith row internally and also the jth column entirely. So if I consider some thread i, j so essentially what is it doing?

It is essentially accessing row i let me say row i1 for matrix 1 and this is matrix 2 and this is matrix output and column j of matrix 2. Now let us look at entity ij + 1. So for that I would have another thread so what is the row and column that this thread would be accessing. So i, j + 1 as to be sit in the same ith row side by side with ij right so thread which is responsible for computing the ith row j + 1th column element of output matrix if I call it tij+1.

So essentially what it is going to access it is again going to access the ith row of the first matrix and the j + 1th column of the second matrix right. Now let us consider some other entity let us consider the point i + 1j so what are the different memory locations that the thread which should compute for this element will be accessing so let us call this thread t i + 1, j. So essentially as we can see so the first one accessed ith and jth the second once as accessed ith and j + 1th column right.

And for this so I am essentially doing an access of i + 1th row and the jth column right okay. So let us again take an another example just for completeness I have then consider 4 nearby locations here. So i + 1th row and j + 1th column right so then I would be having this new thread which as been launched for i + 1th row and j + 1th column that would be accessing for first matrix i + 1th row and for the second matrix the j + 1 the column right. Now see what is really happening so what if the total for computing this 4 elements what is the total number of row's and columns that we have accessed okay.

We have accessed 4 total number of rows and columns that is to be specific 2 rows and 2 columns. But as we can see there is lot of common access so for an example the ith row has been accessed separately by 2 threads who have been computing for this position as well as this position. So the ith row has been accessed by both this threads similarly the i + 1 th row as been accessed by this thread and this thread right.

So I would say that a performance loss why because 1 thread has already accessed the data from the global memory. Now again I am asking another thread to access the same data the same location from the global memory why for doing the computation of a different position. Now of course one way we do that some of this will saved in the cache well we are talking about GPU whether cache's are really small right.

So considering that we are looking into performance our programming for very large matrixes for saving some regret which would mean I am kind of having this duplicated accesses from the global memory this will be a huge performance issue. Why because every global memory access is extremely costly order of magnitude costly. So the warps which will essentially access this global memory would stalled for a lot of time right.

So if I am trying to do the matrix multiplication in a nice way this will definitely not be the good algorithm for doing that. So again just to give an example again so essentially as we can see the ith row would be accessed by I mean if we trying to speak in terms of some hard numbers so the ith row is getting accessed how many times over all in the computation. Of course for in total in the output matrix there would be n elements in the ith row using this naive algorithm I am bringing elements from the ith row into the memory so many times right n number of times by separately by each of the threads.

So that does not make sense it would have been much better if data brought in from the global memory would have been useful for the computation by some other thread the. As you can see that each thread is bringing its own row and own column from the global memory but if it was the case that whatever the thread i brings threads tij brings can also be used by the threads tij + 1 now that would have been much nicer right.

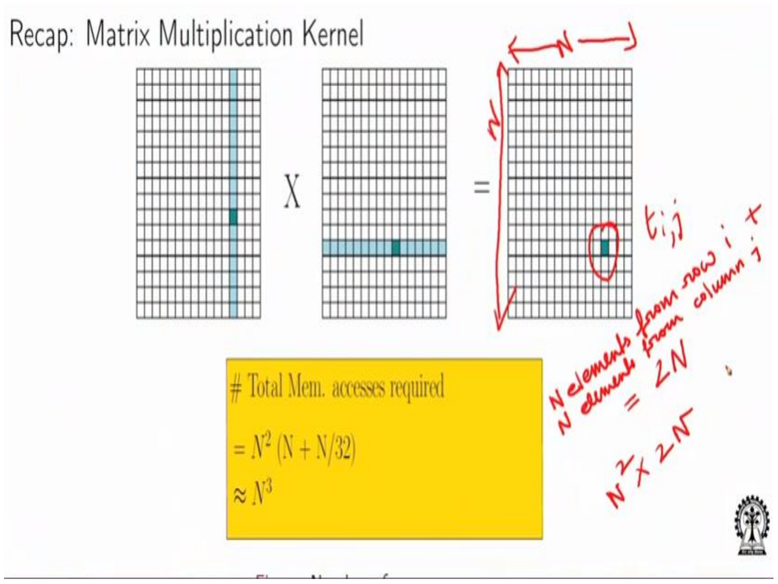
Because then both of them bringing the data of the same location multiply times does not really make sense. So but the question is okay they may not be doing the computation at the same time the tij and ti + 1 j ti +1 or tij+ 1 thread tij and thread tij + 1 may not be active together at the same time. So in that case although since they are not active together in the same time where will the data be resident right.

So that also is an important thing right so essentially for this purpose we exploit the shared memory that is located in the same in the streaming multi-processor. As we know that the shared memory is preferring a consistent view to all the threads inside the block hence threads operating inside the block can actually collaborate among each other let me put it in other way around. The threads operating inside the block can bring in data in such a coordinated manner that the data required by one of the threads in the block is also useful to some other thread inside the block if they are brought from the global memory and they are made resident in shared memory location.

Why? Then each of the threads inside the block would be accessing the data from the shared memory location and that access will be much more faster. So the point to be noted here is we should try to minimize the re-fetching of data multiple times through global memory loads by different threads for doing the similar activity on different data points from the but using the same locations from the global memory the alternate option is make the thread working inside the block collaborate among each other by bringing the data from the global memory and putting the data in the shared memory.

Why because as we have already discussed that the shared memory view is consistent for all the threads inside a block. So they can actually use the data that means for doing their own part of the matrix multiplication computation and while doing that they instead of doing global memory loads they will be actually doing the load data from the shared memory which is much faster. Because the shared memory is on chip and it is located inside the SM much near to the scalar processors or the SP course.

(Refer Slide Time: 15:34)



Recap: Matrix Multiplication Kernel

Total Mem. accesses required

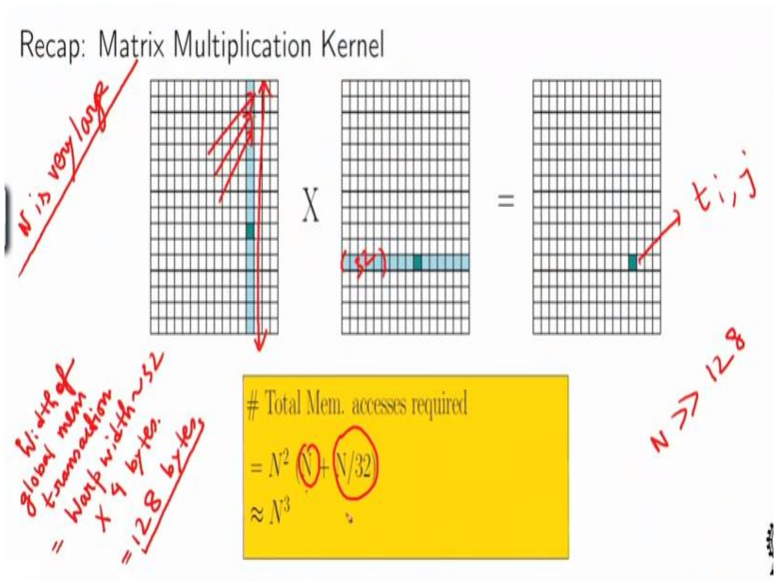
$= N^2 (N + N/32)$

$\approx N^3$

So what would be I mean just consider the picture flit here so what would be the number of memory access is required overall. If I am doing a nice matrix multiplication computation well I mean if I do not consider the use the global memory coalesced accesses then essentially what is going on for doing computation for each location in the output matrix I need access to 1 column and 1 row of data right.

So I need access to n elements from a row and I need access to n elements from column so that is n + n that is 2n I would definitely the thread which is doing this computation would require access to 2n number of elements from the global memory. So n elements from a row so this is the total number of memory elements that 1 thread would access and overall how many threads do I have? So overall I have n square number of threads so this is the total however the number of elements accessed is not equal to the total number of global memory transactions that would be going on right.

(Refer Slide Time: 17:51)


Recap: Matrix Multiplication Kernel
N is very large
X
=
(32)
ti,j
# Total Mem. accesses required
$= N^2(N + N/32)$
$\approx N^3$
Width of global mem transaction = Warp width ~32 × 4 bytes. = 128 bytes
N >> 128

Just recall out knowledge of memory access coalescing so whenever I am doing access for a column the number of global memory accesses I mean I am considering that n is very large. So since n is very large the access of any column location and the access of the previous or the next column location is never going to be coalesced that means they will never be brought from the global memory by a single global memory transaction why is it so let us recall.

What is the width of the global memory transaction so as we have discussed earlier so as long as if since currently the warp with we are considering for more than GPU's is 32. So the global memory transaction would essentially bring in 128 consecutive by its in 1 byte aligned access again I would just repeat the per byte aligned access like we discussed earlier. So as long as this n is going to be large than 128 bytes this accesses of this consecutive locations in the column will all be separated global memory transactions right.

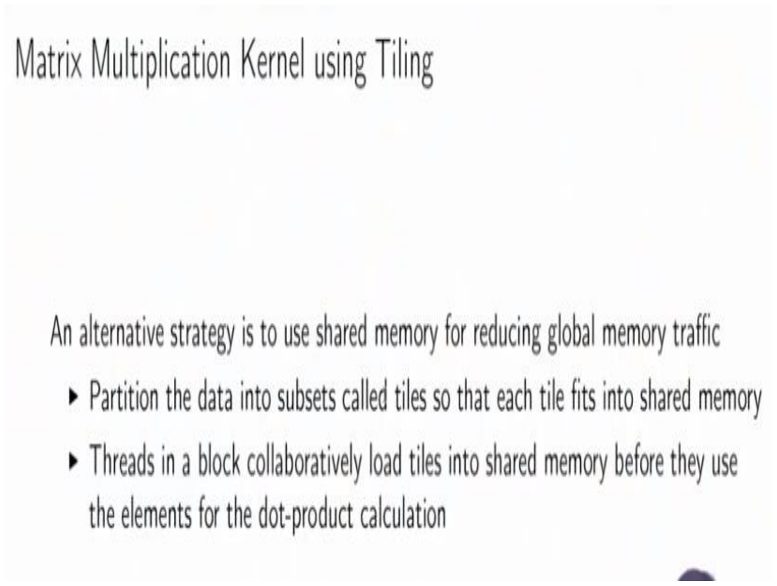
So for computing for 1 location here for n of the column elements I would land up with n global memory transactions however that will not be the case for the row why? Because in the row all this elements in the finally d ram they are sitting side by side right. Because we all know that we are following a row measure kind of sitting in the ram so all of this elements are going to sit side by side they are consecutive elements separated by 4 bytes considering an integer matrix right.

So out of this N the consecutive 42 would be brought by a single transaction right so assuming that this accesses are all byte align that means N is divisible by 32 here the number of accesses we would have is N / 32 right. Now of course if N is not divisible essentially it would be an N / 32 then you take the 4 and plus do one access more something like that. I mean we can also assume here for simply simple purpose it is divisible so I have got this many accesses for 1 element.

So the total number of accesses is N square time this now of course this is the actual figure that is the total number of global memory transaction would be this considering 32 as small if I am considering a very large matrix significantly large matrix I can approximately does (()) (21:36) but of course that we will hold for a very large N is very large okay.

Now we would like to exploit 2 things here first of all the fact that consecutive access in the row are going to be coalesced and the other fact that we have discussing earlier that I can make use of the shared memory threads inside the block and collaborate among each other and in that case many of the global memory loads would be replaced by the shared memory load and that would provide a lot of acceleration in terms of the execution of the code right.

(Refer Slide Time: 22:23)



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

So if I do this optimization then what we get is for an algorithm for doing matrix multiplication using tiling. So in summary we would say that the strategy where going to use is to share do the shared memory which will reduce the global memory traffic. So what will do is we will partition the matrix data space into subsets which are known as tiles in such a way that each tile fits into the shared memory.

So considering that I can fit each tile into the shared memory right I mean that is how we will design the tile size right. So we were discussing earlier the threads inside the block will then collaborative load tiles into the shared memory before they use for the this the elements for doing the dot product calculation here.

(Refer Slide Time: 23:18)

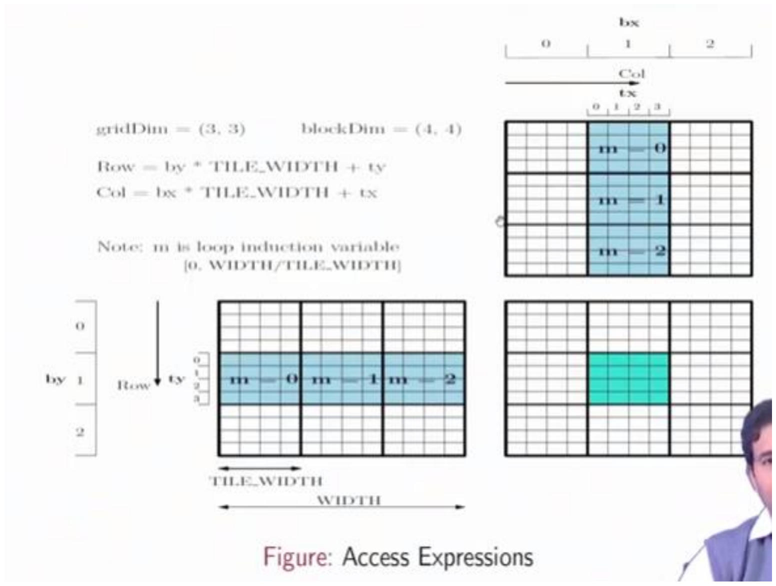


Figure: Access Expressions

So we will that would lead to kind of much more complex algorithm for deciding on that we will talk about that algorithm in the next lecture for that time being let us get comfortable with idea that the shared memory can be used collaboratively using between the blocks between the threads inside the block and I am also going to use the other important parameter here which is that whenever I am doing global memory loads I am always able to coalesce do a coalesce access of the elements in the row with this we will end this current lecture in a next lecture we will go deeper into the algorithm thank you.