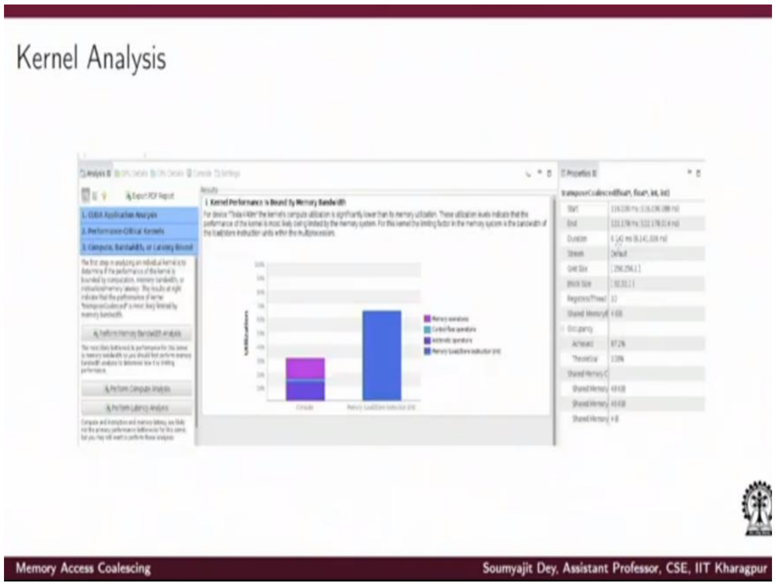
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 06
Lecture No # 26
Memory Access Coalescing (Contd.)

Hi welcome to the lecture series on GPU architectures and programming. In the last lecture we gave a nice example of optimization using the share memory which respect to matrix transpose computation and we saw that how using shared memory you can have very nice global, load and store access patterns and that leads to order of magnitude reduction in the total execution time of the transpose as we saw from our kernel analysis statistics.

(Refer Slide Time 00:54)



Kernel Analysis

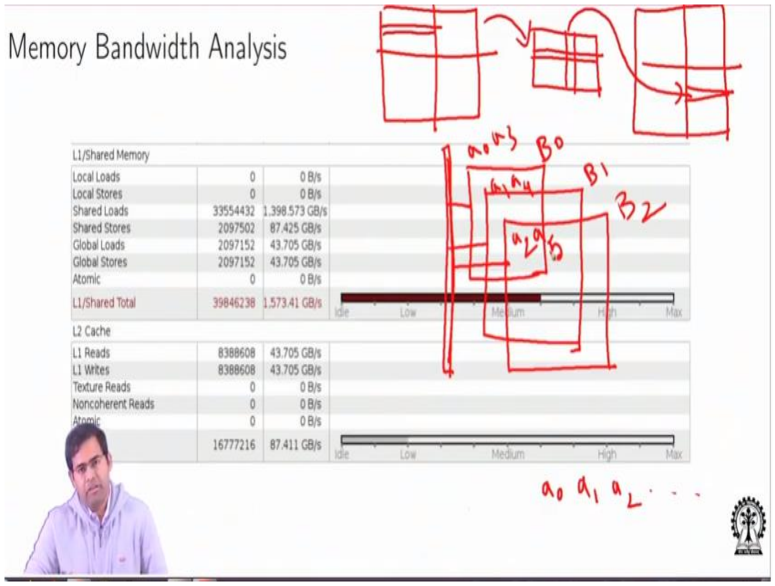
Memory Access Coalescing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

By the way we will just like to repeat that these statistics has been revealed using our system where we have our tesla K40 GPU card. Now of course for if you are trying to have a have results of a of profiling using a NVPROF or some other your favorite profiler for this kind of computation the timing you are going to get of course will depend on the system you will using right.

Here just use this value for comparison among the different optimizations we show because we are running all the different examples or different variants of the same operation in the same system. So consider the relative performance values here.

(Refer Slide Time 01:38)



Memory Bandwidth Analysis

L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	33554432	1,398.573 GB/s	
Shared Stores	2097502	87.425 GB/s	
Global Loads	2097152	43.705 GB/s	
Global Stores	2097152	43.705 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	39846238	1,573.41 GB/s	Idle Low Medium High Max

L2 Cache

L1 Reads	8388608	43.705 GB/s	
L1 Writes	8388608	43.705 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
	16777216	87.411 GB/s	Idle Low Medium High Max

So one interesting thing that we started showing here was that since in our way of computation the shared memory load was not coalesced. So if i just recall our original diagram of how the different locations in the input matrix, output matrix and the tile was getting accessed. So let say this is the tile, this is the output matrix so from one input block one row you are writing here row wise and then you are loading the data from the tile column wise and writing in the flipped block location row wise right.

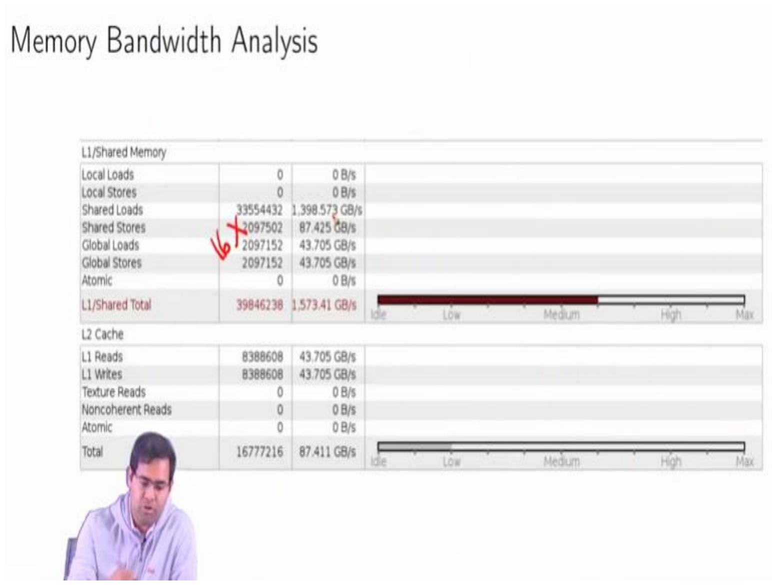
So this loads from shared memory are not coalesced and that would create bank conflicts in the shared memory. So what is a bank conflict? The shared memory is not a contiguous piece of memory element rather it is arranged in set of banks. So in modern GPUs we as we know that the GPU families the NVIDIA defined is what they call as compute capabilities right. So since we are working on a device with compute capability 3 and above we have a share memory with 32 banks.

So essentially what is the share what is the idea for memory bank. Now a memory is arranged into this different sets let me redraw this picture in a nice way. So the memory when I say that the memory segment is bank is not that I have a one big large chunk of physical memory. Rather than that the physical memory is split into smaller chunks of physical memory and they are all accessible in parallel by the hardware right.

So let us say this is the read write box and you can access all this banks in parallel. What is the good thing about this when you read or write data into the shared memory let say this is bank 0, this is bank 1, this is bank 2. Since you can access them in parallel and if your arrangement of data let us pick out some array it is a0, a1, a2, like that there consecutive location right when you are reading or writing the data consecutive value will be distributed across banks like this right.

So when you are writing the data consecutive location gets distributed now the amount of data if you are writing if you same as the number of banks then there is no conflict in terms of the writes. So what do I mean by conflicts?

(Refer Slide Time: 04:50)



Memory Bandwidth Analysis

L1/Shared Memory

Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	33554432	1,398.573 GB/s
Shared Stores	2097502	87.425 GB/s
Global Loads	2097152	43.705 GB/s
Global Stores	2097152	43.705 GB/s
Atomic	0	0 B/s
L1/Shared Total	39846238	1,573.41 GB/s

L2 Cache

L1 Reads	8388608	43.705 GB/s
L1 Writes	8388608	43.705 GB/s
Texture Reads	0	0 B/s
Noncoherent Reads	0	0 B/s
Atomic	0	0 B/s
Total	16777216	87.411 GB/s

Think in this fashion that since I have a warp of size 32 and we devices of compute capability of 3 or higher. I have got 32 memory banks so at the entire warp of thread if they are actually going to store data or load data to or from the shared memory in consecutive location all those locations going to map to different memory banks right. Because I have 32 memory banks, 32 consecutive locations suppose I am loading data from this 32 locations using a warp of 32 threads I will simply access all of the banks in parallel and I will load the data right so that will be 1 load.

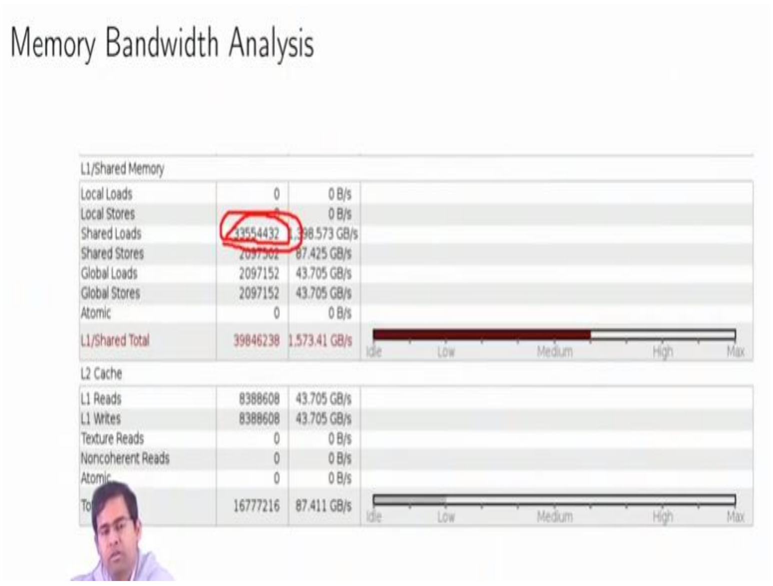
But consider the situation that when I am doing the shared memory loads in our case. So I am loading the data from a column in the matrix and column in the tile size is 32 cross 32. So essentially the column will get what we call as a 32 way conflict because the entire column is located in one bank right. So then the question comes that how much overall number of loads would happen. In this case in this specific case we see that it is almost a 16 x time required right.

Now this as we have discussed with people in NVIDIA forum this is something that is architecture dependent. Although this is a 32 way conflict that we are getting still the memory banks have got a read write speed with which they can actually read or write 64 bits per cycle. Now since in this case our data is each of size 4 bites. So although the conflict is 32 way the number of loads are increasing by the factor of 16 because the data size is half with respect to whatever I can load from the bank using its bandwidth.

Since in this case for (()) (07:06) it is 64 bits per cycle that means 8 bytes in parallel now what does that mean to be more specific suppose that I am working with 8 byte data. If I am working with 8 byte data let us say double procession floating point. And then when I am trying to load such a column of data, the data sizes are 8 byte with all the column all the data positions located in the same bank. I have a 32 way conflict and then since the data width is equal to the banks band memory band width per clock memory band width that is the number of bits that I can access for clock cycle.

So in each load operation I will load one instance of the data. So in that case I will have 32 x increase in a number of loads. In this case it is 16 x right. Some other interesting activities as you can see that since we have got nice access patterns with respective global loads and stores. So both of them are small there is no increase based on the problem that we have been discussing.

(Refer Slide Time: 08:14)



Memory Bandwidth Analysis

L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	33554432	1,398.573 GB/s	
Shared Stores	[illegible]	87.425 GB/s	
Global Loads	2097152	43.705 GB/s	
Global Stores	2097152	43.705 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	39846238	1,573.41 GB/s	Idle Low Medium High Max

L2 Cache

L1 Reads	8388608	43.705 GB/s	
L1 Writes	8388608	43.705 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
To	16777216	87.411 GB/s	Idle Low Medium High Max

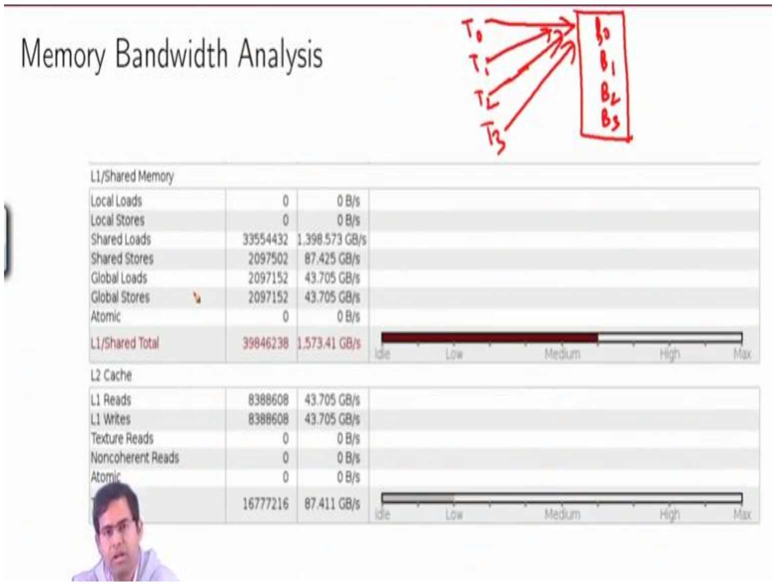
So this is the good thing about the shared memory and also we have pointed out about the bad thing due to bank conflict I have got a huge increase in the number of shared memory loads right. Now how do I decrease this because this again is leading to some loss of (()) (08:35) whether the question is whether it is possible to some how rewrite the code in a such a nice way that this issue of shared memory bank conflict can resolve.

Again I will just summarize this idea of bank conflict which is similar to memory access coalescing so in a global memory I can access consecutive location. Similarly when I am and if a warp can access consecutive 32 locations in parallel and so if that is the amount of data that is required. It can be fetch from the global memory one global memory transactions similar for writes.

Coming to shared memory it is arranged in banks and consecutive locations consecutive data points are written by the threads in a warp in a consecutive bank. So if I have 32 banks and suppose I am going to write 32 consecutive data points it will be written consecutively in across the banks. But unless it is there it is such a regular access pattern. For example suppose I am trying to write data in a column wise way write in a read or write in a column wise way that means suppose all the threads in the warp are trying to write for the tiles column that means or read from the tiles column.

That would mean both read or write operation I am accessing in the same bank. And that gives rise to 32 way conflict. Now this 32 way conflict is also something I will like to explore a bit. So what do I really mean by this. So for that let us take an example for some other cases of conflict.

(Refer Slide Time: 10:17)



Memory Bandwidth Analysis

L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	33554432	1,398.573 GB/s	
Shared Stores	2097502	87.425 GB/s	
Global Loads	2097152	43.705 GB/s	
Global Stores	2097152	43.705 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	39846238	1,573.41 GB/s	Idle Low Medium High Max

L2 Cache

L1 Reads	8388608	43.705 GB/s	
L1 Writes	8388608	43.705 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
[illegible]	16777216	87.411 GB/s	Idle Low Medium High Max

So consider some other threads so let us say thread id 0, thread id 1, thread id 2, thread id 3 like that and let say this is my set of memory banks B0, B1, B2, B3 trying to write data in a column wise way per write in a read or write in a column wise way that means suppose all the threads in the warp are trying to write for the tiles column that means or read from the tiles column that would mean in both read or write operation I am accessing the same bank.

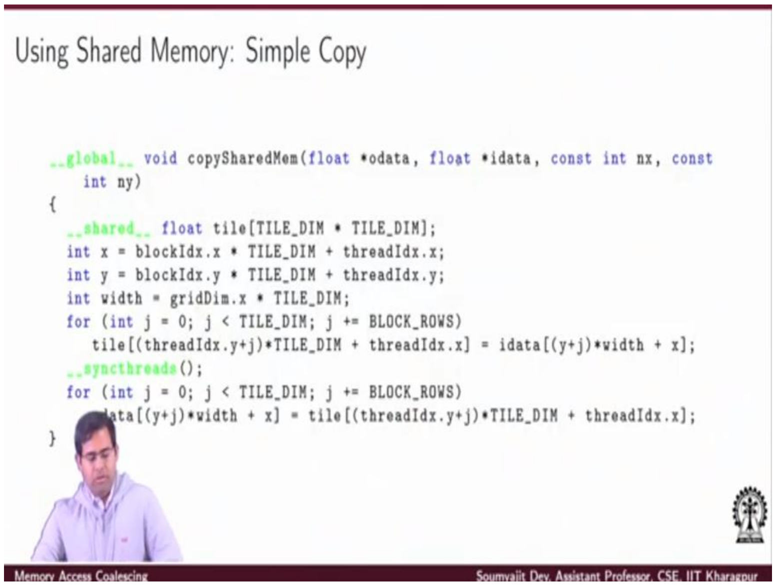
And that give rise to 32 way conflicts. Now this 32 way conflict is also something I really like to explore a bit. So what do I mean by this? So for that let us take an example for some other cases of conflict. So consider some other thread so let say thread id 0, thread id 1, thread id 2, thread id 3 like that and let say this is my set of memory bank B0, B1, B2, B3. Now if the access pattern is like this regular then there is no bank conflict right is where already discussed.

Now consider some other access pattern for example let us say the threads access in stripes of 2 right. So T1 and T2 access a separated by 1 hop sorry T0 and T1 access are separated by 1 hop right. And then let say T2 and continue this pattern T1 and T2 access is again separated by 1 hop right. And then T2 and T3 access its separated by 1 hop right. So let us consider this pattern right. Now let us understand what is the number of bank conflict I get.

Since I have got 4 threads accessing 2 banks right. And for each bank I am looking for 2 transaction, so this make a 2 way conflict right. Now consider this pathological scenario that every bank is considering to access data or from bank 0 like happened in our example that we are accessing the tiles column and that would mean the tile width is equal to the number banks. So if I do the modular mapping then all the data in the column will fall into the same bank and that would give rise to this scenario. So considering the whole number of banks here I have a 4 way conflict.

Similarly this problem we have a 32 way conflict. So this is how the idea of the share memory conflict builds on.

(Refer Slide Time 12:55)



Using Shared Memory: Simple Copy

```
__global__ void copySharedMem(float *odata, float *idata, const int nx, const
    int ny)
{
  __shared__ float tile[TILE_DIM * TILE_DIM];
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] = idata[(y+j)*width + x];
  __syncthreads();
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     data[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x];
}
```

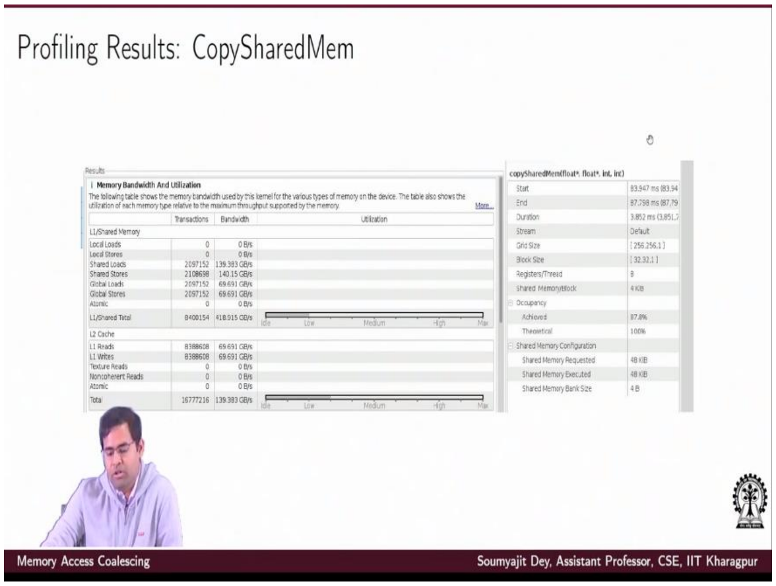
Memory Access Coalescing   Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now just to get some idea that ok suppose I am just have I will I resolve find out some way to resolve the bank conflict. And I want to compare that with the normal share memory copy operation. Before we get into the way we do a share memory bank conflict resolution let us look at some performance trigger with respect to a simple copy operation. So you have a input matrix you access that input matrix using the idea of tiles.

And then I mean so as you can see the similar code that we have again where we continue with our definition of tiles and blocks. I mean I can have a blocks continuing I mean which are addressing a multiple tiles or 1 block 1 thread block contains the same number of threads as in a tile both are fine in in one case the loops will be active and in other case the loop will only iterate ones. But what is going on here? This is a simple copy kernel that means the data is copied tile wise right input data gets in to a tile and then that tile full of data will goes into the corresponding output matrix right.

So this is the simple accesses we are doing here. Now we want to compare this simple copy kernel with the situation of optimization where we are doing a bank conflict resolution.

(Refer Slide Time 14:23)



Profiling Results: CopySharedMem

Results

Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	2097152	139.383 GB/s	
Shared Stores	2108698	140.15 GB/s	
Global Loads	2097152	69.691 GB/s	
Global Stores	2097152	69.691 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	8400154	418.915 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	8388608	69.691 GB/s	
L1 Writes	8388608	69.691 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	16777216	139.383 GB/s	Idle Low Medium High Max

copySharedMem(float*, float*, int, int)

Start	83.947 ms (83.94
End	87.798 ms (87.79
Duration	3.852 ms (3,851,7
Stream	Default
Grid Size	[256,256,1]
Block Size	[32,32,1]
Registers/Thread	8
Shared Memory/Block	4 KiB
Occupancy	
Achieved	87.8%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

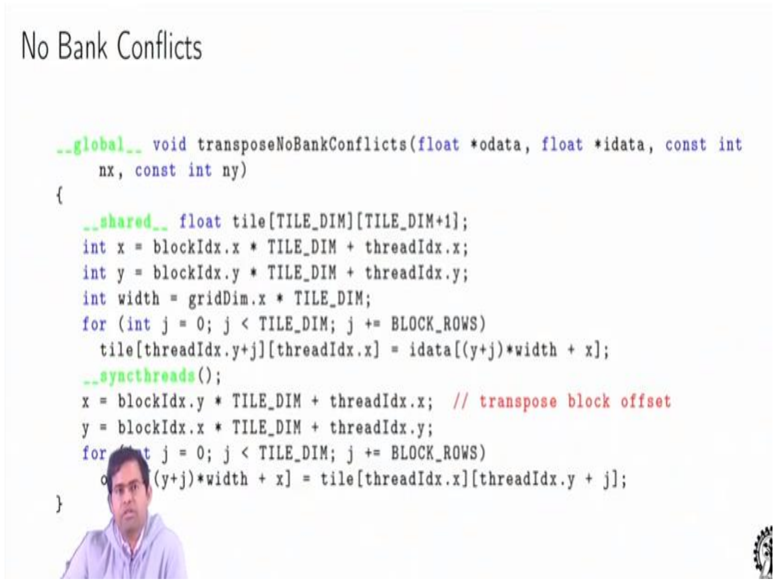
Memory Access Coalescing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So first of all we make an observation that if I am just doing memory copy from one input matrix to output matrix what is the time taken if I am using shared memory. As you can see it is some 3.8 milli 8 milli seconds which is much smaller if I compare it with my results on shared memory based transpose which was 6.1 for 2 right. In both cases we are using shared memory but I can attribute this difference due to this bank conflict why?

Because in both cases my global memory loads and global memory stores are all coalesced. Here also as you can see that the global memory loads and stores are all coalesced along with shared memory loads and stores right. So my problem with the code where I am used the shared memory but I compute with the transpose was that. The shared memory are bank conflict while doing the loads for writing in to the output error. So how do I resolve this bank conflict is the primary problem we want to solve it here.

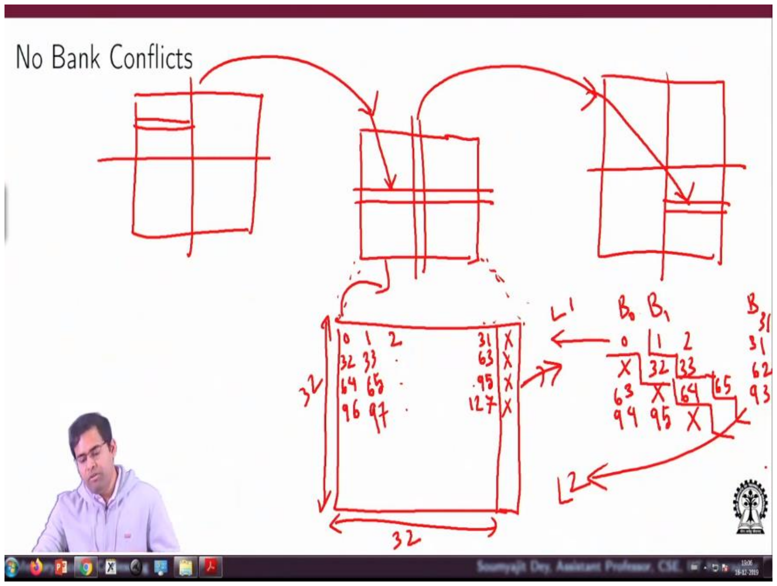
(Refer Slide Time 15:27)


No Bank Conflicts

```
__global__ void transposeNoBankConflicts(float *odata, float *idata, const int
    nx, const int ny)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];
  __syncthreads();
  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

So the idea is very simple keep the code exactly same as it was increase the tiles dimension by 1 in 1 direction. But the question is how does this really help ok.

(Refer Slide Time 15:44)


No Bank Conflicts

So let us try to figure out that what is the effect now if I increase the tiles dimension by 1. So this are access pattern that we have already discussed but now we are trying to figure out that for this tile when I draw a magnified picture of tile so this dimension in the x direction has been increased by 1 right. So let us try and figure out first of all I just like to repeat we have just increase the dimension nothing else has change in the code.

Which means I do not really use this extra location for copying data by any of the threads or I do not really load anything from this location and write back somewhere. So this is the last column which I do not really access I do not write here I do not read here right. But I use the rest of the 32 cross 32 locations for whatever earlier operations I was doing basically all the same operation right.

So when I have written the data here let us understand for one block I have done a row wise access and return the data row wise right. That would mean the threads and the corresponding locations if I just write the ids of the locations, they are getting mapped nicely like this. I do not write anything here and that is how it continues right. This is what is going on right. Point is now what happens to the arrangement of the data in the banks.

Now in the x dimension of that tile I am mapping all this locations to 32 banks and I have 33 locations to map per row right. So in terms of the banks how really is this mapping happened. So this data would go to the data for the location 0 goes to bank 0 like this up to bank 31. And then this location is mapped to 0 right and that was has changed. So now if I try to map this layout in terms of the shared memories actual lay out.

So let us rewrite the bank locations B0, B1 I am just writing the shared memory banks now right. So I have up to B 31 right. So I load the 0 as data first data second data then up to thirty first data and then I have a do not care value in B0 then the thirty second data in B1, thirty third data in B2 like this up to 63 sixty third location right. And then now sorry so since the thirty second location is in B1 so what do we have here it has to be the sixty second location.

Please understand this, this is very crucial. Since I have putting a do not care here so now your B0 starts getting mapped from B1 location right. So sorry here it should be shifted to 62. But then where does 63 goes, 63 would now get mapped in B0 and then after that there was a do not care right. I am removing this it was initially drawn just to understand the banks but since I have physical picture just to avoid the confusion, we removed this.

This is the enlarged view of the tile and this is a corresponding mapping in terms of the actual share memory banks. So sixty third data point is in bank 0 after that we have this do not care. So then 64 data point is then in bank 2 now if I continue like this so this is in B2. So what should we have here? Here we should now have the ninety third data point right. So that would bring the ninety fourth and ninety fifth data points in banks B0 and B1 and here you get a do not care.

Now consider what happens to the load instructions. So I am doing a load by column right. So now I when I am trying to access the data look at the loads that will happen right. So in the first load I get all the data right so essentially first of all let us point out some data points that I want to load. Assume that I want to load from this column right. Because just to keep consistency let us go back to the original code and see what we were really loading.

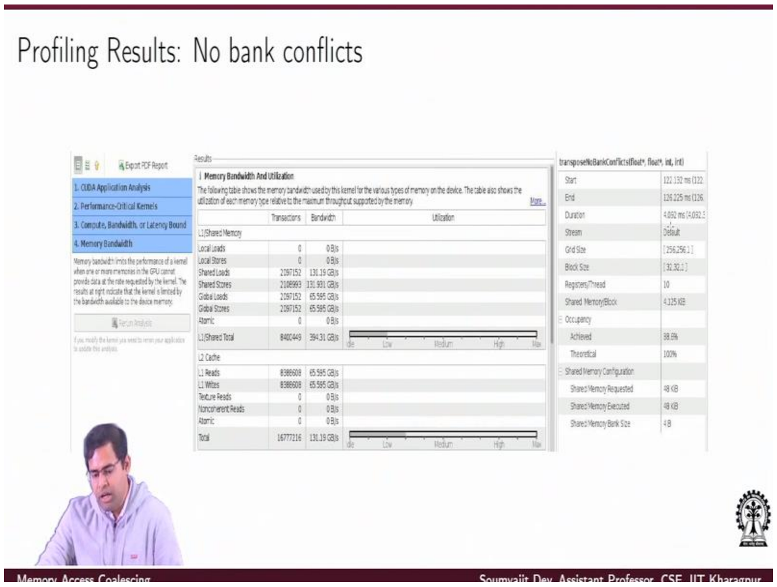
This is the code without bank conflict and it is just that I am are loading from a column right. So as a representative column let us just take a first column which is going to which was in our original implementation where I did not increase the width of the tile. All these data was going to bank 0 right. 0, 32, 64, 96 all were going to bank 0. But now what is the location as you can see 0, 34, 64 like this it is all now distributed right across the banks.

So you have 0, 32, 64 and like this up to the n everything is distributed across the bank they all get out with the load 1 right. Similarly in the next what do I get? In the next column what do I have? 1, 33, 65, 97 if you look 1, 33 here you will have 65 and similarly like that so this is the output that you will get in load 1. And in the next axis you are going to get all the data in the second load and so on hence so forth.

So now due to this insertion of an extra dimensional extra column in the tile I have got memory padding in the shared memory and this memory padding is actually removing the conflict that was happening in each load instruction of the column in the tile when I look at that axis in terms of banks I am always accessing the all the banks parallelly. So I do not have any conflict by loading the values for column from the bank shared memory.

So this work like magic but it is only that simple all you would do is you modify the tiles definition by doing a padding of an extra column. It changes the entire arrangement in the bank. So every column of the data actually is now diagonally distributed in the bank and due to the distribution you have that column uniformly distributed across different banks and they can all be accessed using a single load instruction.

(Refer Slide Time 25:23)



Profiling Results: No bank conflicts

Export PDF Report

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound
4. Memory Bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel. The results at right indicate that the kernel is limited by the bandwidth available to the device memory.

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Results

Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	2097152	131.19 GB/s	
Shared Stores	2108993	131.931 GB/s	
Global Loads	2097152	65.595 GB/s	
Global Stores	2097152	65.595 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	8400449	394.31 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	8388608	65.595 GB/s	
L1 Writes	8388608	65.595 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	16777216	131.19 GB/s	Idle Low Medium High Max

transposeNoBankConflicts(float*, float*, int, int)

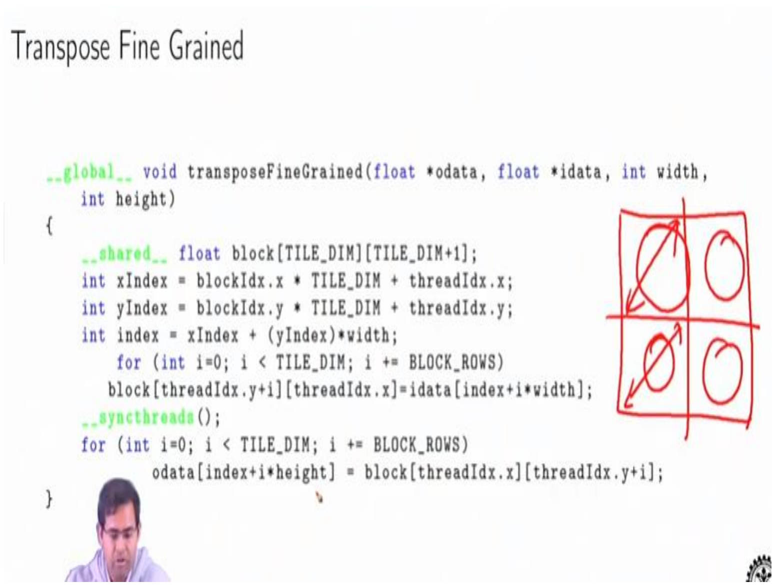
Start	122.132 ms (122.
End	126.225 ms (126.
Duration	4.092 ms (4,092.3
Stream	Default
Grid Size	[256,256,1]
Block Size	[32,32,1]
Registers/Thread	10
Shared Memory/Block	4.125 KB
Occupancy	
Achieved	38.8%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB
Shared Memory Bank Size	4 B

Memory Access Coalescing   Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So that is the wonderful optimization in I would say with that we have very reduced execution time as you can see it is almost 4.1 milli second and the all the shared load, shared stores, global load, global store all are coalesced right. And as you can see this is almost similar in execution time it is only 3.85 with the normal copy operation using share memory right. So here you have successfully resolved all the bank conflicts and your code executes with very minimal execution time and it is almost comparable with a normal simple copy operation without doing the transpose right.

This is how if we understand the memory hierarchy correctly we can remove bank conflicts and do better optimization with respective data.

(Refer Slide Time 26:14)



Transpose Fine Grained

```
__global__ void transposeFineGrained(float *odata, float *idata, int width,
    int height)
{
    __shared__ float block[TILE_DIM][TILE_DIM+1];
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index = xIndex + (yIndex)*width;
        for (int i=0; i < TILE_DIM; i += BLOCK_ROWS)
      block[threadIdx.y+i][threadIdx.x]=idata[index+i*width];
    __syncthreads();
    for (int i=0; i < TILE_DIM; i += BLOCK_ROWS)
            odata[index+i*height] = block[threadIdx.x][threadIdx.y+i];
}
```

With this we will like to end before that we would like to revisit some other example. If you remember the way we spoke of transpose we did a 2 level optimization in the transpose right. The first computation we did was just to change the x and y in the axis. In a such a way that you do not completely flip the x and y and y and x. But you compute the x and y for the original transpose kernel using shared memory such that they represent the flipped block location.

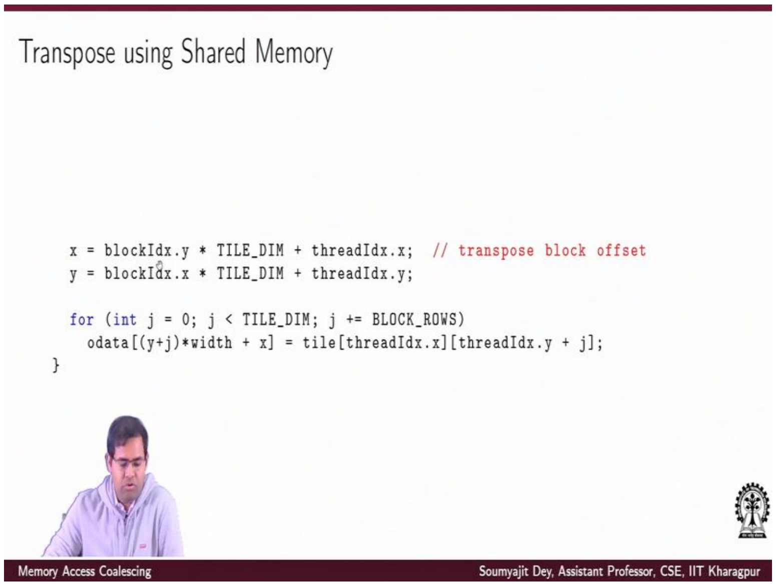
And then in the flip block location you flip while reading from the tile and the reason for doing that was you achieved very nice patterns with respect to read and write. But you did a overall matrix transpose. Now if we require that you do not need the entire matrix to be transpose you only need to do a tile level transpose. What initially I mean is so suppose you have this entire matrix you do not want to change the relative position of blocks you just need to do local transpose here inside the tiles. You want to flip inside the tiles right.

So that would mean all you do is you do not do any re computation of the block ids. But rather you access the data I mean from the input load to a tile and then you read from the tile column wise and write to the output data, without doing a intermediate flipping of blocks it will just repeat. If you remember optimization which is here so you said 2 level thing right. First I flip the blocks and then I flip the location of read from the tile consider this is not done.

So you do not flip the block. So you are reading and writing from the same block but while reading from shared memory tile you have flipped that means you read column wise. So that would effectively do a fine grain transpose as we call it here. So what essentially I mean is I do not I do a local transpose. That means I do not change the block position but inside the same block I am now reading it in a different way and writing it back.

So I am effectively just inside the same block I am doing the transpose operation. So you compute the index to read if you are standard and then you read the data I your definition so here defining block as the thread I mean thread at the tile using the name block here and then from this you write using the flip pattern that we discussed earlier. Suppose if we do not want to do this but you want to do a block level transpose that means you just flip the location of the block but you do not change the internal arrangement of data.

(Refer Slide Time 29:35)



Transpose using Shared Memory

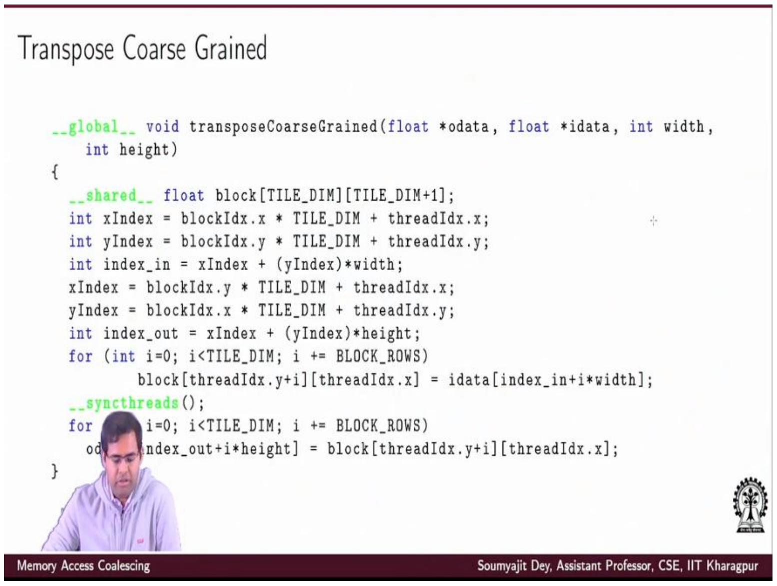
```
    x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
    y = blockIdx.x * TILE_DIM + threadIdx.y;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
      odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Memory Access Coalescing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Again if I go back here so if I do not do this operation then I do not flip the position of the block. Suppose I do this operations but then I do not do this flipping here. I do a normal read write what would that mean I flip the location of the block by this, but then I do the normal read write. So that means inside the block relative offsets of the location do not change. So that is what we can call as a coarse grain transpose.

(Refer Slide Time 30:04)



Transpose Coarse Grained

```
__global__ void transposeCoarseGrained(float *odata, float *idata, int width,
    int height)
{
  __shared__ float block[TILE_DIM][TILE_DIM+1];
  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)
          block[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
  __syncthreads();
  for [illegible] i=0; i<TILE_DIM; i += BLOCK_ROWS)
    od[illegible]ndex_out+i*height] = block[threadIdx.y+i][threadIdx.x];
}
```

Memory Access Coalescing

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So for fine grain transpose we have an execution time has given here and then for the coarse grain transpose as we discussed so as you can see I computed the index in first that is how to read right. And then I flip the block locations using the original code segment I show, and I use this flip block location to compute the index out where to write and then I do a normal read write.

I read from i data using index into the block or the tile and then from the tile I use a normal access as you can see there is no flipping while reading and writing from the tile. You straight write to the tile you straight read from the tile. You use index in to read from index i data, you use index out to write to odata. Now the difference is index in the original usual consecutive location index.

But when I use the index out I have flip the blocks using the first part of the switching. But I am not using the second part of the switching. Here I do not read by column I read by array read by row sorry I ready by row from the tile. So this is the coarse grain transpose that mean this block comes here and some relative location of data remain same with respect to the block right. So this are also associated operation that you can look into and we also provide the execution time and other things of that with this we would like to you can actually look into this data points try and compare with this we like to end this lecture thank you.