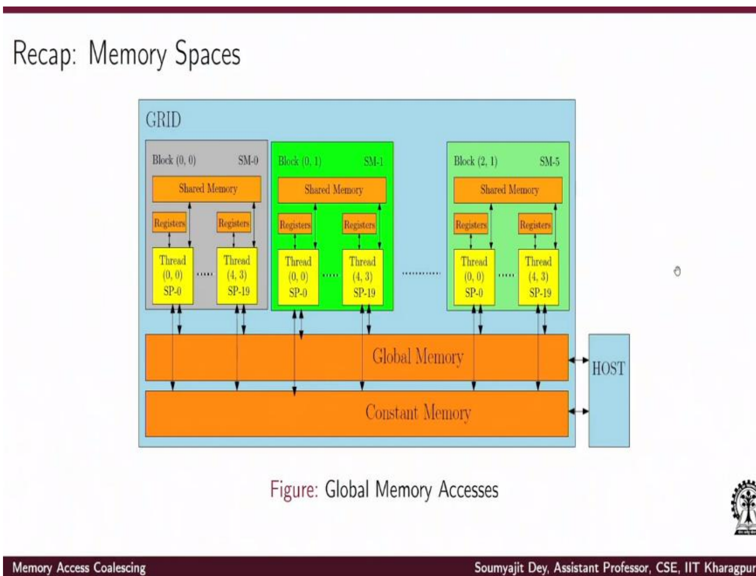
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 04
Lecture No # 19
Memory Access Coalescing

Hi so in the last lecture we have been discussing more about how the GPU warps are formed and they are scheduled inside the SM's. In this current lecture we will be focusing on one of the videos important topic relevant to our optimizing GPU programs and this is memory access coalescing.

(Refer Slide Time 00:51)



So before getting into the detail of that let us first do a small recap on the hierarchy in the GPU memory. How the different memory segment are organized and how really the global memory is accessed by the computing threads inside an SM. So as we have discussed earlier when a kernel launches essentially a grid of thread blocks are launched. Multiple blocks can reside inside one SM as decided by the high level scheduler of the GPU and it depends on the size of the blocks.

1 thread block cannot be split across the SM and so when the set of blocks get mapped to set of SM's I mean in this example we are just showing single block mapping in to single SM like that. so essentially I have a set of threads inside the blocks which are computing which are being

packed by the low level scheduler of the SM into the units of computing called thread call warps. And each of these warps perform progress they are actually progressed to the instruction sequence automatically.

So that I mean in a the all the thread inside the warp are executing in lock step with respective to the instructions. So here we are just shown this earlier picture actually that how each of the SP's are the scalar processor are getting mapped to the threads and they are executing the threads. And of course we will have scenarios where we have more number of threads rather than the number of SP's. So that it would require another low level of scheduling of the threads on the SP's and that is how things are done as discussed earlier.

Now when ever the threads while executing an instruction they will require access to the global memory so they would definitely require 2 kinds of access. They would need to bring operands from memory and they would need to write back updated variable values to the memory. So this essentially should be happening through the hierarchy of the GPU the memory hierarchy of the GPU.

So every access should ideal I mean finally all the variables have their locations in the global memory and also while you have written the program if there are certain constant factor in your program. You may have mapped them also into the constant memory for faster access right. So the when a thread is asking for a variables value it may be it may possibly residing in the L1 cache or it may be I mean the L1 cache part or the shared memory part or I mean of course if you are lucky that it may if it may be residing in the register file segment allocated for the thread.

Otherwise the thread has to access this value from the global memory. Now of course by standard computer architectures techniques as we know a registered access is first, shared memory access is bit slow with respect to the register. But its the global memory access which is a lot slower because essentially this is outside the SM units outside the entire GPUs competent environment it is in the DRAM chip of the GPU.

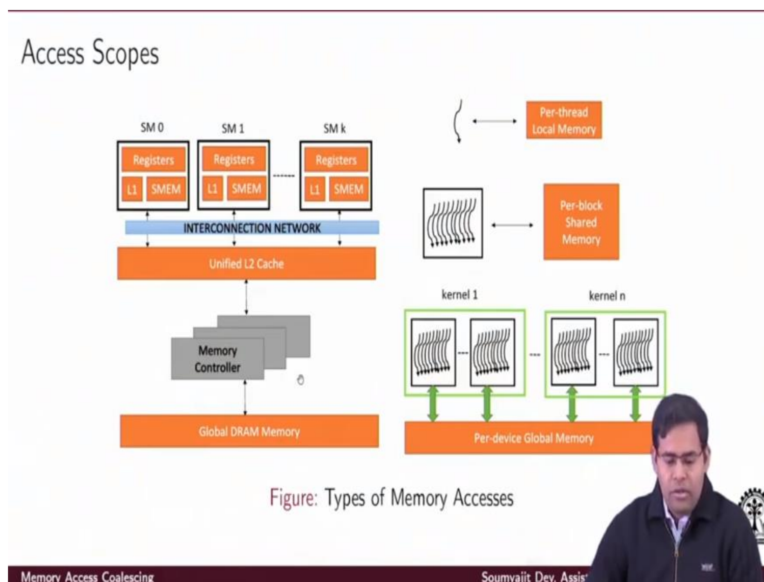
So accessing the global memory is an order of magnitude slower. So if you are writing a program which will be doing lot of access to the global memory essentially your program performance will be low it will not execute with high throughput. If you can somehow read write the program

so that effectively the number of global memory accesses are getting reduced, then your program will perform much faster.

So essentially the important thing here is how can I reduce the number of times the global memory is going to be access by a program. Now this question leads to a lot of memory related optimization some of which will be touching throughout this lecture sequence here. Now of course you can understand that if we are going to reduce the number of accesses of the program makes the global memory we need to some how optimize the accesses and make data resident in the shared memory and the register file.

The shared memory or L1 cache whichever way it is getting configured as discussed earlier. And so we have to optimize the program in a such a way maybe at source level.

(Refer Slide Time 05:21)



So first of all lets redo the scope of computation for the different threads. So you have got this different SM's. Each of the SMs have got their each own as we discussed earlier. Each of the SM's have got their segment of register which are dedicated for them. Each of the SM's have got the memory partition the internal memory partition in to the L1 and the shared memory segment. And then to their interconnection network you have access to unified L2 cache which accesses the bank global memory to the memory controller right.

So for each thread in with doing its computation it has got its own separate private memory segment allocated in the global memory which is also called the global I mean the allocated in the DRAM which is known as local memory for the thread. For every thread block you scan as a programmer define and amount of shared memory which is available to a thread block for doing its computation.

So since it is shared memory all the threads executing the block will have a consistent view of the variables in the shared memory which again means it thread i and thread j belonging to the same thread block are executing and thread i makes an update on a variable it is immediately visible to the some any other thread j this updated value is immediately visible. Since it is in the shared memory.

And of course you have multiple kernels and if I mean in case you are executing concurrent kernel which is the feature supported by modern version of CUDA. So they take you have that would actually mean multiple kernel executing together and they can have their part device access to the global memory.

(Refer Slide Time 07:05)

So now coming to the important question of the latency of this accesses. Now they definitely differ for different memories spaces as we discussed earlier. So this is the nice summary here that the global memory which is accessible by all the threads is the slowest definitely it is the DRAM. The shared memory which is in divided into each of the different SM's it is the

accessible by all the threads inside the thread block it is much more faster order of magnitude faster which is referred to the global memory. However if your variable is resident in the registers then it is access time will be the fastest. So this is the order which is standard like any computer architecture.

(Refer Slide Time 07:44)

Warp Requests to Memory

- The GPU coalesces global memory loads and stores requested by a warp of threads into global memory transactions.
- A warp typically requests 32 aligned 4 byte words in one global memory transaction.
- Reducing number of global memory transactions by warps is one of the keys for optimizing execution time
- Efficient memory access expressions must be designed by the user for the

Memory Access Coalescing

Soumyajit Dey, Assist

Now the question is how really is the global memory access by the threads. We have already discussed that the threads are packed together what we call as warps right. So when this warps are executing the GPU wherever is the threads inside a warp are looking for data to be assessed from the global memory the GPU tries to gather the accesses together and form specific access specific in mean I mean instead of accessing warp thread try to merge the accesses together into global memory accesses so and this is known as coalescing.

So GPU through coalesces is the global memory loads and stores depending on whether threads inside a warp are looking for data seating consecutively into the memory or not. Summary I would just say that the global memory is coalescing the GPU is coalescing the global memory load and stores. Now this load and stores requires CUDA by the warp I mean the threads inside the warp.

And so if I have multiple threads inside the warps which are looking to access global memory location which are consecutive they would be packed together into a single global memory transaction. So of course you have to define what is the global memory transactions? What are

the data points that can be brought together by a single global memory transaction and all that? So suppose you have a warp where all the thread I mean you have 32 threads and the 32 threads I mean you are the warp is executing the load instruction or a store instruction.

And in general let me call it a transaction. So essentially you have got 32 memory request issued in parallel right. If this request are actually going to the for consecutive position in the data space that means you have 32 align 4 byte words which have been requested. Now this is exactly what is defined as a global memory transaction. So a global memory transaction is essentially transaction in which from the global memory you can bring 32 cross 4 number of consecutive bytes from the global memory.

So that is the width of a global memory transaction. When you are accessing the DRAM chip definitely this is how the hardware hierarchy is optimized. The hardware designers know that the global memory access is slow. So he is trying to provide you provide the program or an optimization that okay whenever you access mean access is slow but I am allowing you to access lot of data points in parallel by give you a wide bus.

What is the width of the bus? It is a 32 cross 4 bytes that would mean if in a warp you have 32 consecutive thread ids and they are making a loaded request for 32 consecutive memory location. Essentially you have to do a single global memory transaction. So this is the most fundamental and important thing here. Since you are global memory access width is 32 cross 4 byte if a warp is looking to access 32 consecutive memory warps that memory locations.

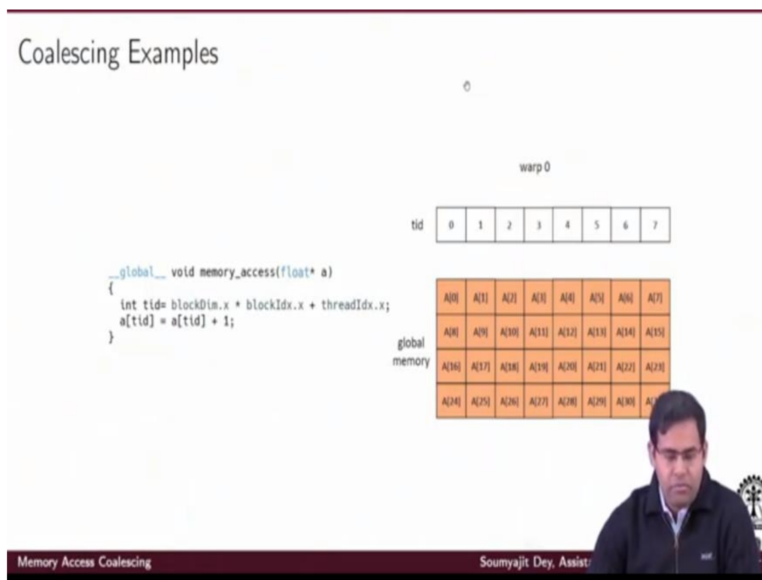
So essentially we are depending each memory element as a 4 byte warp consider let say integers. So you can access the warp can access its corresponding instruction corresponding data points from a memory using a single global memory transactions. Now as we discussed earlier since this global memory transactions are slow overall objective here is to reduce the number of transactions by different warps which are executing across the (()) (11:38).

So for that what is required is you have to you should able to write the program with very efficient what we call as memory access expression. So when ever you are accessing an array position in the memory you have access to expression. So just to give an example suppose you are trying to access M is an array and you have a expression here which is telling that for the

current values of i j and k this is the location you want to access and you put it in some variable a and then the access expression would be this.

So as a programmer you need to define very nice access expression. So that you have to define them in mind in a with a global memory optimization in mind. So that overall when you program executes you have less number of global transaction. So that the warps do not need to wait so much for the memory operations to be done both load and stores. And so that overall progress is much faster of that kernel.

(Refer Slide Time 12:50)



Let us look at some example to make this idea very clear here. So consider the execution of this very simple program. So we have a this small kernel here so is just a memory access kernel. So you are first computing the thread id using this expression standard expression of the block dimension multiplied by block id + the thread id everything is the x in the x direction. So considering single dimensional kernel, single dimensional blocks, single dimensional threads I mean single dimensional arrangement of blocks and single dimensional arrangement of threads inside the block.

So this is your tid and all that you are doing is for every tid you are accessing the corresponding tid in this location in the array a. So let say this is your warp 0 so for this warp 0 you have got the consecutive thread ids and we are just taking an example here. So you have let us this 8 threads

executing inside warp 0. So when this line of the code is executed by all the threads inside the same warp what really happens? What are the memorial immense that are accessed?

As you can see by looking into the access expression so this your expression which says that you just access the memory location corresponding to the tid. So warp 0 contain tid 0 1 2 3 4 5 and like that. So warp 0 is accessing memory location A0, A1, A2, A3, A4, A5, A6, A7 like that right. So I mean considering that this is you going to be a global memory transaction we are not drawing here 32 thread ids and all that just to make the points very clear.

So essentially what are you really doing you can have a single global memory transaction for this read operation. Here it is simplest example again we are assuming that the global memory transaction width is 32 is 8 cross 4 bytes just assume that. We are not drawing 32 tids and all that here. So essentially I have one global memory transaction for which we will suffice for doing the read operation for all the thread ids here right.

Why let me just repeat again because as we discussed that the global memory access is quite wide. Assuming that in this case the global memory access is defined as 8 cross 4 bytes that many bytes 32 bytes I can bring in parallel. So essentially for this specific case I can bring in all the data points required for executing this line I can do the corresponding load operation here by the single global memory transaction.

So essentially the global memory transaction would be defined based on the width of the warp. As we have discussed already since standard warp which is 32 the global memory transaction can be done in such a way that you can bring in 32 cross 4 consecutive bytes from a memory together right. So assuming here for our simplistic example I mean we have this many threads and you are bringing in the corresponding data points. You can bring all of them together in a single transaction.

If I just want to generalize this all that I would like to do is put 31 here. So this would go up to 31 this next would start from 32 and this would go up to 63 and like that. We are just taking a simplistic example here. We are just considering that the warp width is 8 and similarly the global memory transaction is also 8 cross 4 bytes right. So here we can see that since all the access are consecutive location in the memory for the threads executing inside a warp.

I can have 1 transaction for doing the read and then there will be an execution of the increment operation and then again there will be one transaction for updating all of this location by another write operation for the global memory through which all this location will be updated in parallel because again the location are consecutive. So what happens if the program is a bit different? So just continuing with the animation for this program you warp 0 executing and bringing everything together with 1 global memory read followed by one updating everything here in one global memory write. Then again bringing together all the data points to one global memory read and again updating everything from this using one global memory write. Similarly for warp 2 read and write and that is how this will go on.

(Refer Slide Time 17:56)

Now consider a small difference here in the program. So let us look into the earlier program once again so earlier program was this atid is atid +1. In this program we are providing an extra parameter here access with offset and the offset value is s. So you compute using normal notion the tid and then what you do is you do the access for atid + s as location right.

So what you are doing is? You are accessing the tid + s as location adding it up and then updating again the atid + s as location for the array a right. So that is what you are doing. Now let see how it goes. So in terms of the reads what is happening consider the s value as 1 now that would mean if I again start looking it to warp 0 with thread id 0 I am doing an access of a1

location right. So earlier it was this the access was this but now the access has kind of changed right.

So you continue like this and tid 7 is looking for A8. So overall with this change what is really happening? You will have the first transaction for locations and in the next transaction you just bring A8. So this together gives you 2 transactions. I hope the point is clear here again I have got 8 threads inside the warp they are looking for 8 data points to load to update and to write back. However the problem is the access here is offset. In the sense that the access is not properly aligned here right.

So when the access would happen for this memory although these are consecutive locations but fundamentally if you look into our definition of the access what did we really want? There has to be 32 aligned 4 byte warps right. So for in this case by warp definition there has to be 8 cross 4 bytes byte aligned access. But the alignment is wrong here right. Essentially here what is happening is you have got an access of consecutive location but the access is starting from A1.

So entire thing is not align with the warp right. So that would actually lead to 2 global memory accesses here. I hope this is clear. So essentially the DRAM bandwidth is utilized with respective to the warps width. If the warps width in this case is 8 so the accesses would be so like when I am defining the access will be aligned at the position where it is basically so the first row can be done through a global memory transaction.

So as you can see in this picture the memory organization is shown with respective to this access alignment right. So the moment I put in offset the alignment is lost I mean so in the first access I can only get things from A1 to A7 because in if I am looking for data for A0 to A7 they all come inside the aligned access but A8 cannot be aligned I mean cannot be brought in with A7 in once access because when I bring A7 the aligned access is do is being in terms of a the warp width times four number of bytes.

So it can bring it will actually bring in data from A0 to A7 from which actually A7 is only useful for me. So for the next thing I will require another access right. That is why when in go for A8 that is different access here. Overall I have two different global memory transaction for the read

operations and again I would have 2 different global memory transactions for doing the write operations.

So just the introduction of this offset I mean of course the offset introduction is not I mean it may be required right depending on the program that you want or you have a desirable property for the program. If this is what you really want as you are access pattern, then you have the following problem. So essentially the number of global memory transactions earlier it was for this entire operation for executing this line of code every warp was spending 2 global memory access.

But now instead of 2 you are spending 4 global memory accesses right. So immediately you have lot of slow down in terms of execution time of each of the warps. So I hope this point of access alignment now becomes clear with this example of offset. So just to summarize again we showed the earlier example where all the accesses are nicely align because we have the warp width times 4 bytes.

So that is the definition of the global memory transaction and that is essentially one of the important points here that the warp would request 32 aligned 4 byte warps. So one transaction each of the transaction would happen in spaces of this 128 bytes considering warp width of 8, warp width of 32. But whatever is the warp width in your system multiply that by the requirement of the integer storage that is 4 bytes.

So that is basically the unit of global memory transaction. All the global memory transaction are essentially aligned with those with that amount of width here. So whenever I am doing this transaction I am getting everything in a single transaction of the global memory. What ever is the requirement here from A0 to A7 because they have a nicely aligned access here. However whenever I have a offset based code then comes a problem here right.

Simply because the accesses have to be aligned with the warp width. So here in this case for this kind of arrangement I have the first global memory access for the warp 0 can only I mean it cannot form a global memory transaction starting from here and looking up to this. They have to be aligned at the warp width times 4 number of bytes. That is why it will have to make two transactions the first transaction will bring in this many data.

Out of which this many would be useful. Again the first transaction would bring this many data out of which only this many would be useful. The second transaction would actually bring in this entire row out of which 8 would be useful. So 2 global memory transactions. I hope that provides the summary of this thing. But overall what we can see immediately what is the performance loss here and this is how it will keep on going on. Now consider a different type of access pattern. So in my earlier access pattern it was all with respective to offset. I am doing a (()) (26:52) here.
(Refer Slide Time 26:54)

Coalescing Examples: Strided

warp 0

tid	0	1	2	3	4	5	6	7

```
__global__ void strided_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid*s] = a[tid*s] + 1;
}
```

Misaligned strided access: s=4

4 global memory transactions for read
4 global memory transactions for write

global memory

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]
A[16]	A[17]	A[18]	A[19]	A[20]	A[21]	A[22]	A[23]
A[24]	A[25]	A[26]	A[27]	A[28]	A[29]	A[30]	A[31]

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

Now we consider what is defined as strided access. So this is another kind of access pattern which we often see in our programs. Essentially all I am doing is there is another integer parameter s and whenever your parameter accessing the locations in the memory you are multiplying the thread id with s. We can see such patterns in very simple I mean normal programs that we write. So this is one of the patters that is a strided access. What would it really mean?

Consider again our simplistic example with warp warp width 8 and global memory accesses happening in 8 times 4 byte width. Of course and those access are aligned with this width. So consider that s value is 2, now if s value is 2 I can see that for warp 0 the total number of accesses are actually spread across 2 rows here in our picture right. So for the thread ids 0 to 7. I am going to actually access location A0 I do not want any access location A1 because thread id

1 is going to access A2, thread id 2 is going to access A4, thread id 3 is going to access A6 and like that and that is how it continues.

So in this case also if s is equal 2 the number of global memory transaction required for executing this a entire instruction for entire program line would be that I require 1 read and 1 write for the with respect to the program. But it will actually trans pack to two global memory transaction for the read and the two global memory transaction for the write operation. But what happen if I increase this consider s is equal to 4. Now things are further catastrophic.

For warp 0 again considering 8 threads inside the warp look at tid 0 it is looking for accessing A0 or what about tid 1? Since s is equal to 4 it is going to access A4. So overall now the access is spread across this entire memory block right. So for the 8 strides each of them are accessing a position shifted by 3 from the earlier thread right. So you are essentially accessing A0, A4, A8, A12, A16 like this up to A28.

So essentially since the access that again aligned with those with the restriction for warp width times 4 number of bytes. That is this much is your access alignment. So essentially for covering this entire segment of the global memory you would need how many? You would need 4 global memory transaction for the read and 4 global memory transactions for the write. So as you can see this is from our performance perspective for the program this is going to be a impediment I would say.

Because whenever you are going to a strided access you need more number of global memory transactions for both for read and write and strided access is something that we often we do in our normal programs right. So that is not something that we can do away with. So we have to figure out ways in which the program may be written in an alternate way such that I can create a equivalent program but the access are intelligently managed.

So this are the important performance in impeding factors whenever we are writing code which is going to which mean which I mean for the for the GPU and you have to write programs where you have the variant of the program which actually minimizes this access or may be it helps in coalesces the access. That means the program is such that the warp actually a is looking for accessing consecutive memory locations which can fit in inside the single memory transaction.

This is known as memory access coalescing. With coalesced access you can reduce the number of global memory transactions and that is going to be a very important performance factor for your GPU programs with this we end this lecture thank you.