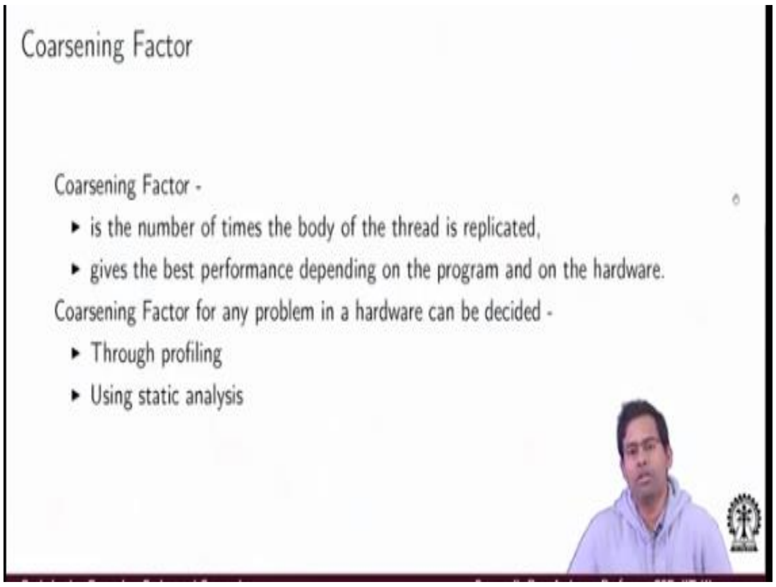
GPU Architectures and Programming

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

Lecture – 39
Kernel Fusion, Thread and Block Coarsening (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming so, if you remember in the last lecture we were discussing thread coarsening as a possible optimization for GPU programs.

(Refer Slide Time: 00:37)



Coarsening Factor

Coarsening Factor -
- is the number of times the body of the thread is replicated,
- gives the best performance depending on the program and on the hardware.

Coarsening Factor for any problem in a hardware can be decided -
- Through profiling
- Using static analysis

And in a short summary, we just discussed that what are the good and bad things of coarsening and we just tried to motivate why beyond a point coarsening threads too much may not be of help and in that regard, we define; we are trying to define rather that what is the notion of coarsening factor so essentially, by factor I mean that how much work I am going to allocate part thread over and above a base line implementation.

So, in that way a coarsening factor is the number of times I am going to replicate the body of a thread that means, the number of times I am increasing the part thread activity with respect to a base line implementation and what is interesting is to figure out what is the best performance, what is the coarsening factor that gives the best performing, performance depending on the program and on the hardware right.

So, it is not something constant, it very much is a function of what program is under consideration and what is the target GPU and of course, I mean it is easy to understand why that would happen because as we have discussed earlier that whether coarsening is going to help beyond a point or not depends on the amount of architectural resources that are available in the hardware.

And up to some point, doing coarsening is fine that you are increasing thread activity without decreasing the occupancy of the hardware at some point, you are going to hurt the occupancy of the hardware but still coarsening may be good because you are utilizing the hardware resources more efficiently because each coarsen thread is making more efficient use of the hardware.

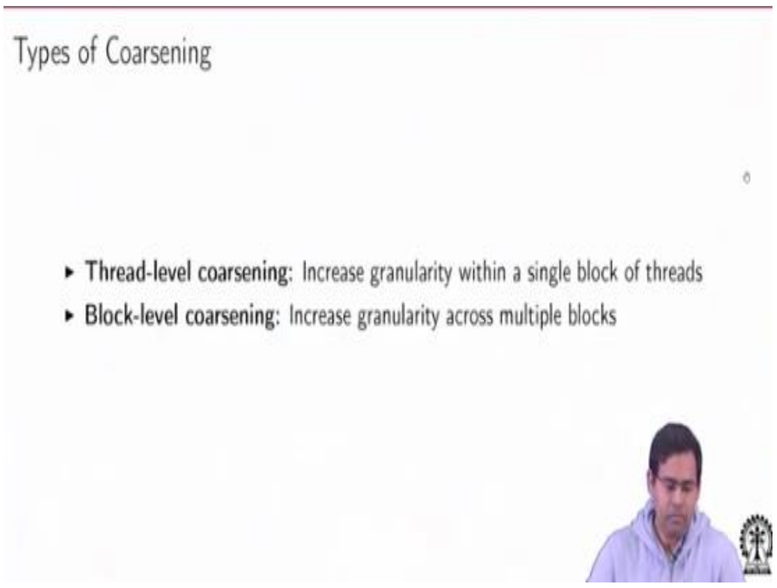
But beyond that point it may happen that the occupancy reduces so much that you lose out on the effective parallelism over the lifetime of the program and whatever is the best coarsening factor is I mean, deciding that statically is; I mean is one way that you can do a static program analysis to figure out a possible coarsening factor, it may not be the best but you can figure out a possible coarsening factor.

And also the other way would be that you try different coarsening factors and it is really an intricate problem because the architecture will have lot of parameters, the program can have lot of dependences, so figuring out the perfect coarsening factor, you can try doing that using a static analysis and you can without any guarantee that it is basically, the best coarsening factor.

The alternative can be that you try out different possible coarsening factors, profile, each of the implementations, different coarsen implementations in the architecture, profile them, profile each of the coarsening factors for multiple possible input runs and create some speed-up analysis and figure out what is working best for you. Now, the static analysis based methods which can give you a good coarsening factor would be sound ones.

That means, they would give you, they will tell that well this coarsening is feasible to implement without hurting the functional equivalence of the program but it may not be able to say that whether that indeed is a best possible coarsening or not.

(Refer Slide Time: 03:49)



Types of Coarsening

- **Thread-level coarsening**: Increase granularity within a single block of threads
- **Block-level coarsening**: Increase granularity across multiple blocks

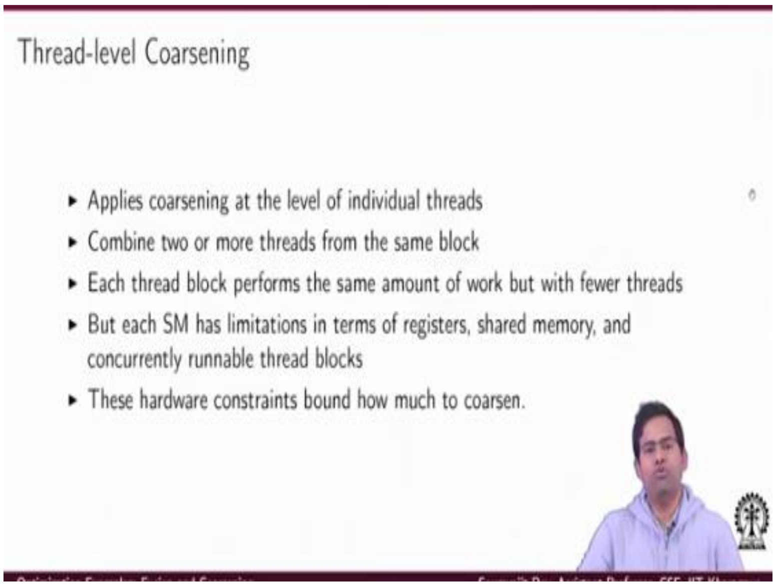
So, the different types of coarsening just like we discussed different types of techniques of fusion like what are the different possible ways in which you can implement fusion among GPU kernels, similar to that there are different possible ways in which you can coarsen a GPU kernel and basically, you have to increase the parts thread activity now, that can be done at the thread level or at the block level.

So, when we say that you are doing a thread level coarsening that means, increased granularity within a single block of thread, so essentially your coarsening threads by giving them more activity part thread but those are all activities inside a single thread block, the other could be that you do block level coarsening that is you increase the granularity of coarsening across multiple blocks.

That means, when you thicken or coarsen a thread, you give it more activity not from the original block of thread but more activity from other blocks of thread. I hope this is clear let me just reiterate, so when I do thread level coarsening, I will coarsen a thread by giving it more activity but that activity was originally inside this specific thread block, inside who; which I am talking about the threads.

When I say block level coarsening that means, I am coarsening each of the threads inside the thread block by giving them extra activity which is not the activity of the original thread block but from some other thread block.

(Refer Slide Time: 05:19)



Thread-level Coarsening

- Applies coarsening at the level of individual threads
- Combine two or more threads from the same block
- Each thread block performs the same amount of work but with fewer threads
- But each SM has limitations in terms of registers, shared memory, and concurrently runnable thread blocks
- These hardware constraints bound how much to coarsen.

So, let us first discuss thread level coarsening in detail, so when I am applying the coarsening, I am applying it at the level of individual threads and so essentially, I am combining 2 or more threads from the same block, activities of 2 or more threads I am delegating it to 1 thread inside the block. So, each thread block now performs the same amount of work.

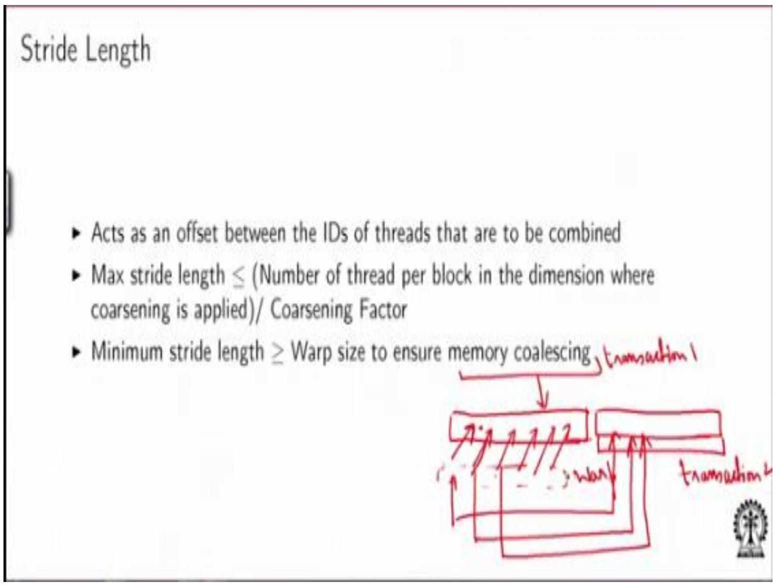
Because I have not delegated work from other thread blocks to threads inside this thread block, since I have not done that so now, each thread block performs the same amount of work but it is able to achieve that with coarsen threads which are fewer in number so, I am now decreasing the threads per block while doing the same functionality of the thread block okay.

But when we do this, each of these trimming multiprocessors have their limitations as we have discussed earlier in terms of the registers, the total register file size inside the SM that amount of shared memory inside this and concurrently, runnable thread blocks like how many concurrent thread contexts that SM can hold, right so, these are the 2 limiting factors and that actually limits the total amount of; so total number of effective threads that will really be launched in the SM after I thicken or coarsen the threads.

Just to make the point clear so, every SM has a upper bound on the total number of concurrently runnable thread blocks right, now it also has an upper bound on the size of the register in the shared memory. So, when I coarsen the threads, I increase the part thread demand of registers and share memory, so due to that the SM maybe also getting further limited in terms of the actual thread blocks that it can run concurrently.

And this hardware constraints will actually I mean, this is something we have already discussed, these hardware constraints will actually decide on how many threads I can really run that will affect the occupancy which in turn will give the bound on how much really I should apply the amount of thread level coarsening.

(Refer Slide Time: 07:27)



Stride Length

- Acts as an offset between the IDs of threads that are to be combined
- Max stride length ≤ (Number of thread per block in the dimension where coarsening is applied)/ Coarsening Factor
- Minimum stride length ≥ Warp size to ensure memory coalescing

So, the next important thing is what is the stride length across which I should do the coarsening right, so this acts as an offset between the IDs of threads that are to be combined so, when I am combining 2 threads what should be the offset between them, right? Now, the maximum stride when I allow, so first of all we are still talking about thread level coarsening but when I am coarsening threads, it is not necessary that I just coarsen tid with the amount of activities for the original tid and the tid plus 1.

But rather I am trying to coarsen the thread with tid, the thread ID tid with activities of original threads with a thread IDs being tid and tid plus s, where s is a stride length right. So, this amount or this stride length can have some limits, first of all what should be the maximum stride length; it should be less than the number of threads per block in the dimension where the coarsening is applied divided by the coarsening factor.

So, let us try and understand what it means so, let us consider that this is the total thread arrangement and I am using a coarsening factor C equal to 2 right, so effectively I am going to use these many threads right, I mean half of this number of threads okay, so maybe yeah, so let us say I am going to use half of this number of threads and I am doing the coarsening in this dimension.

And so, when I am choosing the stride length so, the number of threads per block in this dimension, let it be some X right, so divide by 2 right, so that would be the maximum number of threads; thread IDs in these dimension in the coarsen kernel, right. So, the maximum stride length for the threads has to be less than this; that is the limiting factor why because; of course, if it goes beyond these, then I have a problem.

Because the thread seating at the boundary of this coarsening boundary, they those thread IDs plus the stride value would shoot beyond the original thread block boundary, right. So, I if you just consider the number of threads per block in any specific dimension, in which we are applying the coarsening, so let us say this is the dimension we are applying the coarsening, you divided by the coarsening ding factor.

So, that would give you the total arrangement of threads that would be there in the coarsen kernel. Now, when I am talking about that stride length, I hope this is clear the stride length has to be limited by the original dimension in that original threads per block number in that dimension divided by the coarsening factor because if I consider a stride length, which is greater than this, then what happens?

If the thread which has the tid is sitting in this boundary, those plus the stride length would go beyond the thread dimension in this, in the original arrangement of the data, right. So, the maximum stride length will be limited by this equation but at the same time, what should be the minimum stride length? Now, just to make sure, that we do not want to disturb the original memory behaviour of the program right.

So, let us say originally, I had a few tid's which are doing accents of some data sequentially, so that when these thread IDs they get packed inside a warp perform coalesced access of the data from this memory, I do not want to disturb this behaviour right, so let us understand that if I make one thread, each of the threads in this warp, if I am trying to coarsen them by making this thread ID to the job of accessing this thread ID followed by the next let us say.

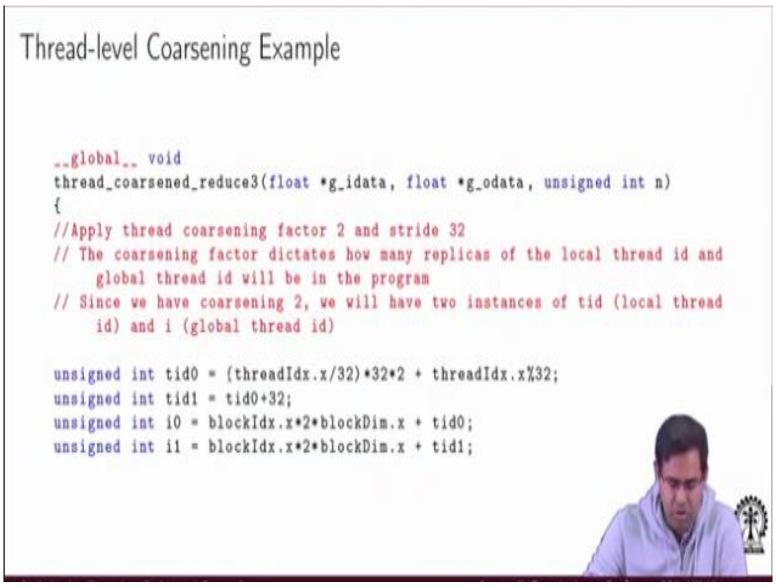
Again, the other thread ID followed by the next and so on so forth, then I may possibly lose out on the memory coalescing. So, when I am trying to coarsen each of the threads, I would like to have the minimum stride length to be greater than the warp size, so that whatever is the behaviour of the threads inside the warp, they do not lose out on their coalesced memory accesses.

But rather when the thread does its coarsened extra activity, it belongs to another separate coalesced global memory transaction, so those would belong to another separate global memory transaction like this. So, this is the original transaction of the threads, this other set of transactions were supposed to be let us say, this is one transaction, this is other transaction. I do not want to disturb this nice behaviour.

So, if I just make the strides greater than the warp size, then essentially I am not disturbing the original memory coalescing, whereas if I do something like I am accessing for each thread ID, I am accessing consecutive locations, that does not make real sense, right because then in many cases it may happen that the original memory coalescing behaviour which was nicely distributed across the threads that may get into a problem.

So, I hope with some examples you can actually work this out that why we would like to keep the stride length greater than the warp size.

(Refer Slide Time: 14:30)



Thread-level Coarsening Example

```
__global__ void
thread_coarsened_reduce3(float *g_idata, float *g_odata, unsigned int n)
{
//Apply thread coarsening factor 2 and stride 32
// The coarsening factor dictates how many replicas of the local thread id and
     global thread id will be in the program
// Since we have coarsening 2, we will have two instances of tid (local thread
     id) and i (global thread id)

unsigned int tid0 = (threadIdx.x/32)*32*2 + threadIdx.x%32;
unsigned int tid1 = tid0+32;
unsigned int i0 = blockIdx.x*2*blockDim.x + tid0;
unsigned int i1 = blockIdx.x*2*blockDim.x + tid1;
```

So, with this as a motivation let us try to look at a coarsened version of our reduction kernel in fact, if you remember that in a reduction kernel whenever we are trying to do access, we are trying to ensure that the global memory transactions are not nicely coalesced and even the shared memory reads are nicely coalesced, right without any kind of shared memory bank conflict.

So, we let us first go through this example, here we are trying to use a thread coarsening factor of 2 with the stride as 32 which is equal to the warp size now, this coarsening factor is dictating how many replicas of the local thread ID and global thread ID will be in the program, right. So, I hope this is clear so, we have to replicate this local and global thread IDs in the program, right.

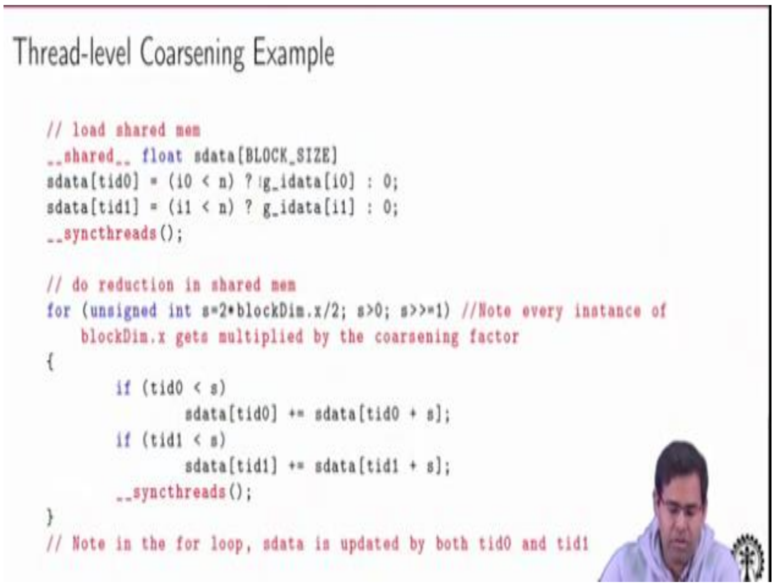
So, since we have chosen a coarsening factor of 2, we need 2 instances of local thread IDs to access 2 consecutive data, 2 possible data values for a given global thread ID right. So, let this tid be the local thread IDs and i is the global thread IDs right, so we will need 2 instances of both of them here so, we compute this tid 0 and tid 1, right. So, that would actually give us this different local thread IDs as you can see that the local; the tid 0 is a standard local thread ID inside the block.

And we know this because we are just calculating the offset right, so we are just assuming that it is all in the; were coarsening in the X dimension here right, since were coarsening in the X dimension with this calculation we are finding out the offset in the local thread ID and then we are doing; since, we are choosing a stride of 32, so that would give me the second location from where I will try to do that thread level activity.

And then I compute i0 and i1which tell me what is the global thread ID corresponding to this local thread IDs, right so, it is just that I know already what is the block, right so, with that block ID and all that I just will add tid 0 and tid 1, observe this multiplication factor of 2 in terms of the block dimension because now, the thing is I am going to access the corresponding data points and I am launching half of the threads.

So, that means that I am going to have half; half the value of effective block dimension while defining the thread blocks, so in order to get the suitable access patterns for the corresponding locations in the data, I will have this multiplication factor by 2 here in the blocks, right.

(Refer Slide Time: 17:33)



Thread-level Coarsening Example

```
// load shared mem
__shared__ float sdata[BLOCK_SIZE]
sdata[tid0] = (i0 < n) ? g_idata[i0] : 0;
sdata[tid1] = (i1 < n) ? g_idata[i1] : 0;
__syncthreads();

// do reduction in shared mem
for (unsigned int s=2*blockDim.x/2; s>0; s>>=1) //Note every instance of
    blockDim.x gets multiplied by the coarsening factor
{
        if (tid0 < s)
                sdata[tid0] += sdata[tid0 + s];
        if (tid1 < s)
                sdata[tid1] += sdata[tid1 + s];
        __syncthreads();
}
// Note in the for loop, sdata is updated by both tid0 and tid1
```

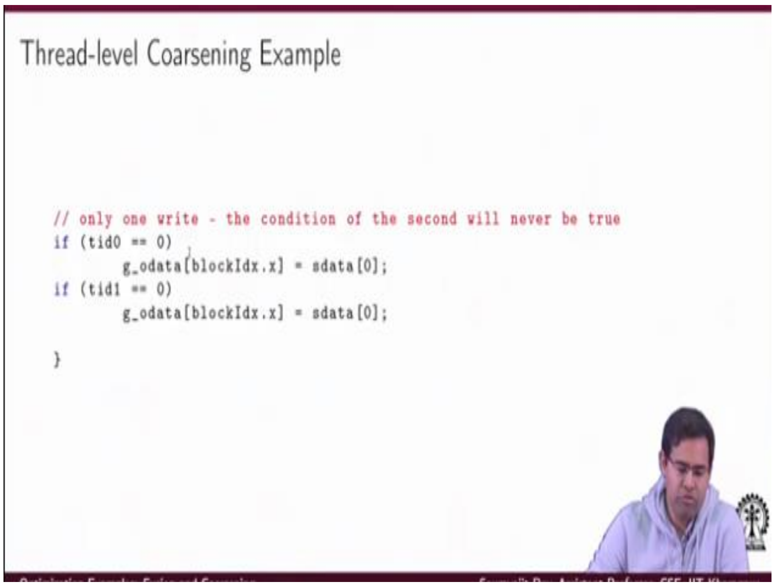
So, with this we are able to compute the i0 and i1 the positions for which I am going to do the global memory accesses for the respective data points. Now, so this is basically the reduction kernel with thread coarsening, so what we are trying to do is that first each of these thread ID; these global memory locations i0 and i1, we are just checking whether their valid locations for this kernel and then we are bringing them into a shared memory.

And once we bring them into the shared memory, we are doing the usual reduction step right, now so as you can see this is the loop for the usual reduction step like we have discussed earlier, right. Now, just observe one simple thing like just like we have done it here since, we have a coarsening factor of 2, so we will have half of the original number of threads in the X dimension, so that is why wherever I have block dimension is getting multiplied by 2, right.

Similarly, here it is getting multiplied by 2, this divided by 2 is the original code semantics for the reduction kernel, right and then inside we have the original code that you just do an addition in shared memory with a stride of s but now you are doing it for, so this is basically this strides of which of the for loop in reduction whereas, our choice of stride for coarsening is 32 which is already hard coded here right, just to avoid the confusion.

So, this s is the effective strides we divided getting divided by 2 in each iteration of the standard reduction kernel right, inside the for loop instead of having one if statement of a standard reduction kernel we have, these are coarsen kernel so, we have 2 if statements for 2 different locations right that is why we have this common. That in the, for loop, the s data is updated by both the tid’s, so by both tid 0 and tid 1, right.

(Refer Slide Time: 19:33)



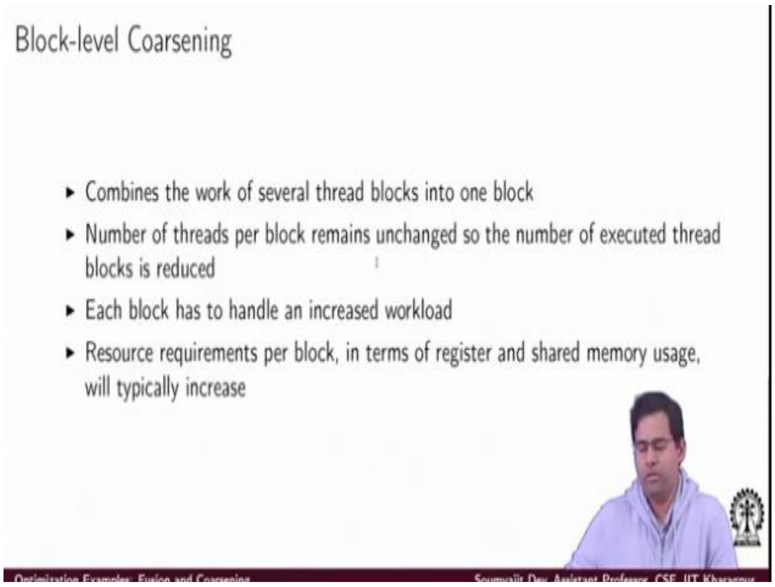
Thread-level Coarsening Example

```
// only one write - the condition of the second will never be true
if (tid0 == 0)
        g_odata[blockIdx.x] = sdata[0];
if (tid1 == 0)
        g_odata[blockIdx.x] = sdata[0];

}
```

And when we do the right, only one of these can be 0 right, so the condition for the second the other will not never be true right, so whichever is 0 for that we will be doing the output data calculation, right I mean, loading back to the global data. So, with this we have an example of coarsening by 2 for the standard reduction kernel.

(Refer Slide Time: 20:03)



Block-level Coarsening

- Combines the work of several thread blocks into one block
- Number of threads per block remains unchanged so the number of executed thread blocks is reduced
- Each block has to handle an increased workload
- Resource requirements per block, in terms of register and shared memory usage, will typically increase

Optimization Examples: Fusion and Coarsening   Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, let us just have a discussion on block level coarsening, so as we saw in thread level coarsening, we were doing the coarsening inside the same block and while for coarsening, we are making a choice of the stride, the stride should be that inside the coarsen kernel, the threads do not access locations beyond the thread block size; reduce thread block size so, it was the original block dimension in that coarsening dimension divided by the coarsening factor that was the maximum possible stride.

And the minimum stride that was definitely greater than the warp size, so that any original memory access coalescing pattern that was a good part of the original kernel should never get disturbed that is why we were not; we are always choosing a stride which was greater than the stride length now, coming to the other part which is the block level coarsening.

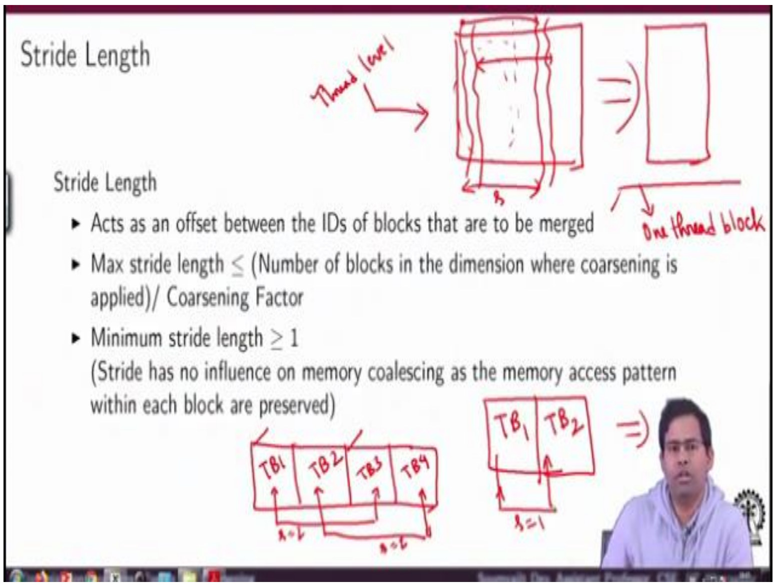
Now, we are I mean, what we are really going to do is; that we will essentially combine multiple thread blocks to one block that means, when I coarsen a thread, the thread should be doing its original activity additionally, it will be doing activity from some other thread from another block, right. So, effectively in the original thread level coarsening, what was happening is the thread number of threads per block was getting reduced due to that coarsened threads.

But here what will happen is the number of threads per block will remain unchanged but we will have the number of; effective number of blocks getting reduced, right. So, each block still has to handle and increase workload as you can see in the original case in the thread level coarsening, the power block activity remains same number of threads per block reduced threads per block got coarsened with original thread original thread blocks activities.

Here, the thread block sizes remains same, the number of; total number of blocks reduces by the coarsening factor since, the thread block sizes remain same and the threads gets coarsened so, each block handles the increased workload. Coming to resource requirements per block in terms of register and shared memory usage, such resource requirements will typically increase.

So, when we talk about resource requirements per block, then in terms of registers and shared memory usage, these requirements typically increase.

(Refer Slide Time: 22:38)


Stride Length

Stride Length
- Acts as an offset between the IDs of blocks that are to be merged
- Max stride length ≤ (Number of blocks in the dimension where coarsening is applied)/ Coarsening Factor
- Minimum stride length ≥ 1
  (Stride has no influence on memory coalescing as the memory access pattern within each block are preserved)

So, what is the choice of stride length, when we are talking about block level coarsening? So, now we are going to discuss the stride in terms of blocks, this acts as an offset between the IDs of blocks that are to be merged, right as you can understand now, we are talking about bringing activities from different blocks together. So, if I am trying to draw a picture originally, when I was doing thread level coarsening, so if this was the original block, I was kind of coarsening threads inside the block by let us say this were the original warps for the threads.

So, now when I coarsen the thread, I am delegating this thread its original activity along with the activity here at a stride length right, so by delegating activities at a stride length together to the original thread, we were coarsening and creating smaller thread blocks, right but now when we are doing block level coarsening, so let us say I have a thread block let us say, this is 1 thread block; thread block 1, this is thread block 2, like that.

So, when I coarsen, I coarsen one thread here with activities from threads in the other block right, now here when I do this, my stride length s is 1, now this is not general, right. So suppose, so here I am not going to threads but I am drawing thread blocks right, so this is one thread block, this is a thread level coarsening. So, suppose these are 4 thread blocks and I pick up a thread from here and I give it activity of the its original activity and some activity from the thread here.

And similarly, for other threads here getting the activities from this that thread block 3, since they are sitting at a distance of 2, so here the definition of stride length is the stride across blocks, right. So, with this we can just say that the maximum stride length can be discussed in a similar way but now at the grid level, right. So, if I am coarsening in a specific dimension, so then the maximum stride length would be limited by the number of blocks in the dimension in which coarsening is applied divided by the coarsening factor.

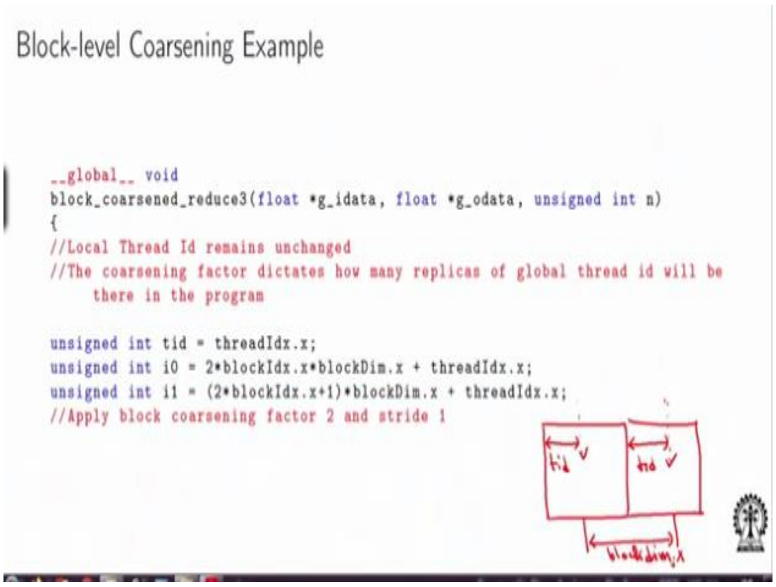
So, just to observe the difference between thread level and block level coarsening, so when we are doing thread level coarsening, the maximum stride length was limited by number of threads in the dimension of the thread block, where coarsening was applied divided by the coarsening factor of the thread block. In this case, the maximum stride length is less than equal to the number of thread blocks in the dimension, number of thread blocks inside the grid of the kernel in that dimension where coarsening is applied divided by the coarsening factor, right.

So, if I say that in this dimension I am applying and the coarsening factor is 2 so, I will be merging this, right and so I am; so, let us say there are 4 thread blocks here and the coarsening factor is 2, so maximum stride length is less than equal to 2 and when I operate with s equal to 2, I am operating at the maximum stride line possible, right and then we have the idea of minimum stride length.

Now, of course that would be 1 because by definition of block level coarsening, I have to pick up threads, thread activity from another block, the nearest would be the next block, so the stride is greater than or equal to 1. Now, since the blocks we are now using the thread blocks for different thread blocks for selecting threads, so try doing have no influence on memory coalescing as the memory access pattern within the blocks.

Or whatever is original memory access pattern inside the block that is always preserved, I mean it is quite easy to understand because when you are forming warps, you are forming warps with threads from inside a block, right and since you are coarsening across blocks, the original warps they are packing their memory access patterns, their memory coalescing whatever was there in as a warp level activity, as a warp level coalescing due to global memory transactions, they do not really change.

(Refer Slide Time: 28:29)



Block-level Coarsening Example

```
__global__ void
block_coarsened_reduce3(float *g_idata, float *g_odata, unsigned int n)
{
//Local Thread Id remains unchanged
//The coarsening factor dictates how many replicas of global thread id will be
    there in the program

unsigned int tid = threadIdx.x;
unsigned int i0 = 2*blockIdx.x*blockDim.x + threadIdx.x;
unsigned int i1 = (2*blockIdx.x+1)*blockDim.x + threadIdx.x;
//Apply block coarsening factor 2 and stride 1
```

So, let us take an example of block level coarsening, so here your local thread IDs remain unchanged, the coarsening factor dictates how many replicas of the global thread ID will be there in the program, so observe the difference; earlier you replicated the global, local thread IDs and then you actually, replicated the global thread IDs but now, your local thread ID is same, you are using.

The reason the local thread ID is same is that the local thread ID gives you a specific offset inside the block, right earlier, you were trying to access 2 different thread level activity inside a single block, so that would mean different possible offsets inside the single block, so you had computed 2 different local thread IDs and of course, their corresponding global thread IDs.

But now, your offset will remain same but for the same offset, you will be accessing different blocks so, for the same offset you are computing 2 different global thread IDs using 2 different blocks. Considering here that the stride is 1 and the coarsening factor is 2, since the coarsening factor is 2 in the X dimension, so you are multiplying block IDx, block Dim x; the block dimension x by 2.

And then you are going to take another thread at the same offset from the adjoining block, so the original block dimension by 2 times block IDx that same thing here, you are applying to the next block, so twice of block IDx in that dimension plus 1, right the adjoining block, in that block you go to the thread ID with the same offset. So, since we are operating at one stride length, so the thread should be separated by one block dimension dot x value, right.

So, maybe with this introduction to the block level coarsening example, we will end this lecture and in the next lecture, we will go into further details, thank you.