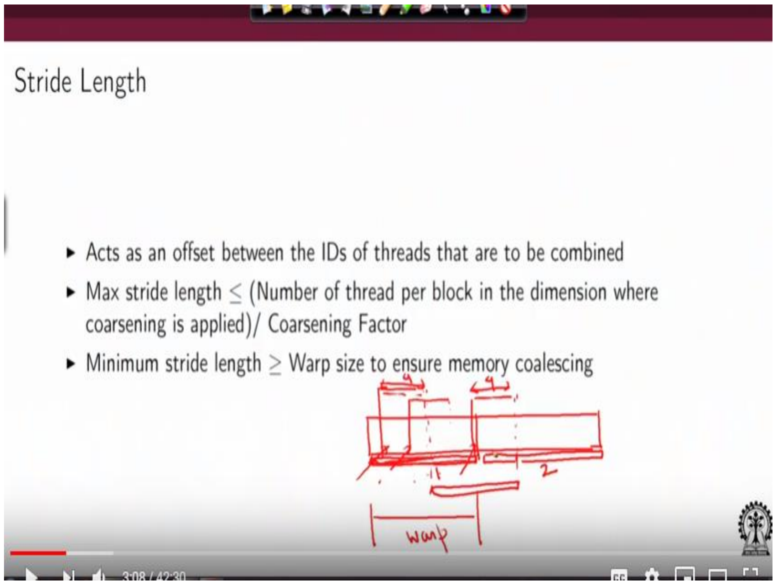
GPU Architectures and Programming

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

Lecture – 42
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 explained in detail the idea of thread level coarsening and we are looking into some examples of block level coarsening.

(Refer Slide Time: 00:39)



Stride Length

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

So just a small point we like to make before getting into further with the block level coarsening example. So if you remember in the thread level coarsening we talked about what is the minimum stride length and we said that it should be greater than the warp size. So just to clarify a bit that I mean of course we understand that the basic idea is that it should ensure whatever was the original global memory transactions and coalescing behaviours they should warps fine.

Just to clarify why we will I mean clarify further on this consider this example that you have a warp of threads. So this is one warp and it was originally accessing a consecutive data elements now you have coarsened them let us say by some factor of 2 and the stride length is not greater than the warp size. The stride length is let us say 4 so if it was greater than the warp size then it the thread should access the original consecutive data elements followed by.

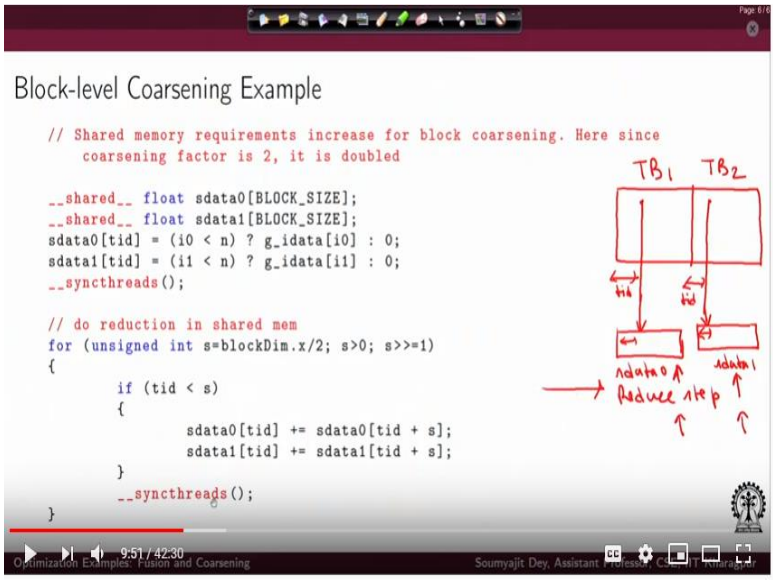
So this is the original consecutive 32 data elements in one global memory transaction and then since if the stride is 32 then it will access the next 32 consecutive data elements in a next transaction that is perfect coalescing activity. Now if the warp size the stride is still 4 than in the first transaction it is bringing the data like this the original one global transaction. But what more does it want each thread now once the data sitting at a offset of 4 right?

So in the next transaction is going to bring the same data starting with an offset of 4 his much right. So in the next global money transaction is going to bring so sorry starting it should the glitch money transaction. So in the original behaviour this was the first transaction this was the second transaction.

But now this would be the first transaction and the second transaction let us say the offset is this offset is this the second transaction would be something like this. Now see there is lot of commonality in the data points. So we are actually losing out on the coalescing behaviour in the global scale looking at all the threads in the kernel because there is lot of reuse happening we are not able to cover more number of global memory elements through a perfect coalescing behaviour that was achieved in the original.

Now this can be easily alleviated if we increase the stride from 4 to 32 which is the warp size because then there will be no extra redundant global memory transaction or bringing data and perfect coalescing would be achieved. So this is one point we wanted to clarify about thread level coarsening and then let us move to the example on block level coarsening we are talking about.

(Refer Slide Time: 03:41)


TB1
TB2
tid
tid
sdata0
sdata1
Reduce step
9:51 / 42:30
Optimization Examples: Fusion and Coarsening
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Block-level Coarsening Example

```
// Shared memory requirements increase for block coarsening. Here since
    coarsening factor is 2, it is doubled

__shared__ float sdata0[BLOCK_SIZE];
__shared__ float sdata1[BLOCK_SIZE];
sdata0[tid] = (i0 < n) ? g_idata[i0] : 0;
sdata1[tid] = (i1 < n) ? g_idata[i1] : 0;
__syncthreads();

// do reduction in shared mem
for (unsigned int s=blockDim.x/2; s>0; s>>=1)
{
        if (tid < s)
        {
                sdata0[tid] += sdata0[tid + s];
                sdata1[tid] += sdata1[tid + s];
        }
        __syncthreads();
}
```

So the first idea about block level coarsening was that since the threads will be picking up activities from blocks sitting at a specific stride length considering coarsening factor of 2 we are going to compute 2 global threat ids in this case its i0 and i1 and they are both corresponding to thread Ids which are sitting at the same offset value of tidx.

So we compute only 1 local thread id and we use that single local thread id to get multiple global thread ids from different blocks since we are using a coarsening factor of 2. So you multiply the block idx by 2 wherever it occurs and since we are using a stride of 1 you should have this 2 block idx+1 for taking the thread from adjacent block right threads from adjacent blocks and threads with the exact same local offset.

So now once I have picked up the threads the rest of the thing is quite same you are going to bring the data from the global memory but then there is something important earlier in the previous example of thread level coarsening you are coarsening the block right you are coarsening the threads by using activity inside the block.

So effectively the requirement of the each block was remaining same right the compute requirement of each block was remaining same the threads inside the block we are getting thickened and the number of threads inside the block were decreasing by the coarsening factor.

But now you are creating you are creating a block with more amount of activity with respect to the original block and you are not reducing the number of threads right.

So when we talk of a reduction kernel or for that matter any kernel the resource requirements that the block level is increasing right. So again I will just like to highlight in the original thread level coarsening the resource requirement at the block level was kept constant the thread level requirements were increasing. So the SM could accommodate more number of thread blocks right in case the part thread level resource requirements were not too high but here what is happening is the thread level activity is increasing but the threads per block is not decreasing its constant.

So effectively the block level activity is increasing for example since I have a coarsening factor of 2 for a reduction kernel. Now each thread is shared inside the block is bringing in effectively double the amount of data and so we will need 2 share memory locations. So we are just allocating to save memories is rate as 0 as data 1 each of size equal to their new block size right and the block size of course is the same because we do not decrease the number of threads per block.

So now for thread id the first thread id I am just using the global threat id i0 to load data in s data 0. I am using the global threat id i1 to load data in the other shared memory which is s data 1. Observe one thing when I am using i0 or i1 I am loading data from adjacent blocks I am loading the data into 2 shared memories.

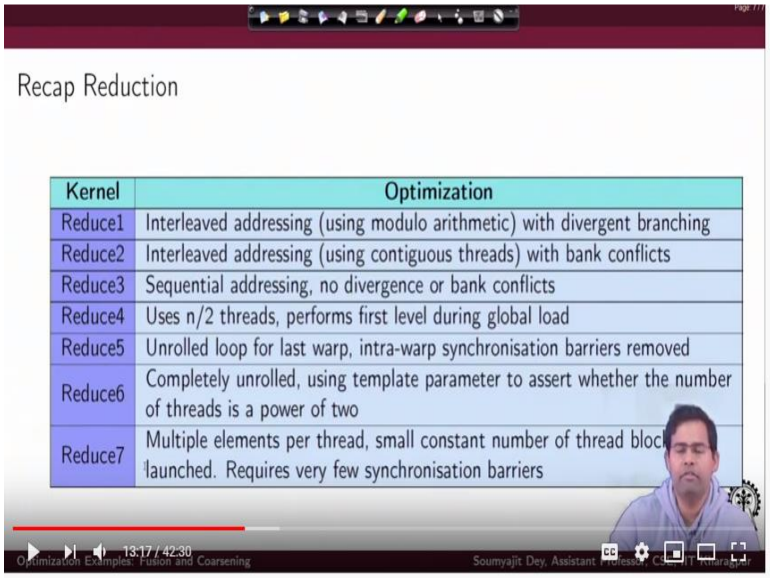
Since I am loading the data from adjacent blocks with the same offset I am loading the data in that different shared memories in the same offset. So just to highlight consider these adjacent blocks so these are adjacent blocks right thread block ,1 thread block 2 let us say I am loading data from here right the offset is tid I am loading data from here, here also the offset is same right.

Now this data will go to shared memory s data 0 this data will go to shared memory s data 1 and the location in which that it I would go will also have the identical offset right. Because this shared memory sizes are same as the thread block sizes right. So with one thread I am loading data from 2 adjacent thread blocks into 2 shared memories 2 different shared memory segments at identical locations okay.

Now once this is done you perform the reduction in each of this shared memories in parallel right. So all the threads would parallel load data into the shared memories then all the threads would go into doing the reduction. So it will do all the threads will do reduced step here as well as here then again in another reduced step parallelly in both the s datas and like that right. So earlier we were reducing in a power block basis and when we did thread level coarsening each thread was reducing more number of locations.

Now we are reducing across blocks and each thread is doing 2 reduction steps in adjacent blocks by bringing the data in the corresponding share memory locations and once all the reduced steps of course keep on synchronizing for each stride value where the stride by stride I mean the stride of reduction in the original for loop.

(Refer Slide Time: 10:15)



Recap Reduction

Kernel	Optimization
Reduce1	Interleaved addressing (using modulo arithmetic) with divergent branching
Reduce2	Interleaved addressing (using contiguous threads) with bank conflicts
Reduce3	Sequential addressing, no divergence or bank conflicts
Reduce4	Uses n/2 threads, performs first level during global load
Reduce5	Unrolled loop for last warp, intra-warp synchronisation barriers removed
Reduce6	Completely unrolled, using template parameter to assert whether the number of threads is a power of two
Reduce7	Multiple elements per thread, small constant number of thread bloc launched. Requires very few synchronisation barriers

13:17 / 42:30
Optimization Examples: Fusion and Coarsening
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So if you just do a small recap of reduction. So these were the different optimizations that we are going one write for so for reduction 1 your optimization was that you are using so there were a lot of problems in that basic reduction kernel. So there was interleaved addressing but you are not really using consecutive threads so the I mean.

So first of all that was the problem because the warps will contain threads which are not really active. So the warps will also have lot of underutilized occupancy of threads that was first problem and then there was the modulo arithmetic which was also overhead and then there was divergent branches right.

Because some of the warps were not that some threads in the warps would actually have no warps right and then you in the reduced reduction 2 step you used interleaved addressing but now you have to actually warps to it continue contiguous threads and that led to balanced warps because the threads will not diverge but still we had the problem of bank conflicts and then in reduction 3.

So this is again I am doing a small recap of the reduction steps because this is relevant here so in the reduction three step what you did was you actually perform sequential addressing and you actually remove the shared memory bank conflicts then in the fourth step of reduction you half the number of threads and delegated some activity to the threads in terms of global loads. So each thread was practically course sent to do global loads.

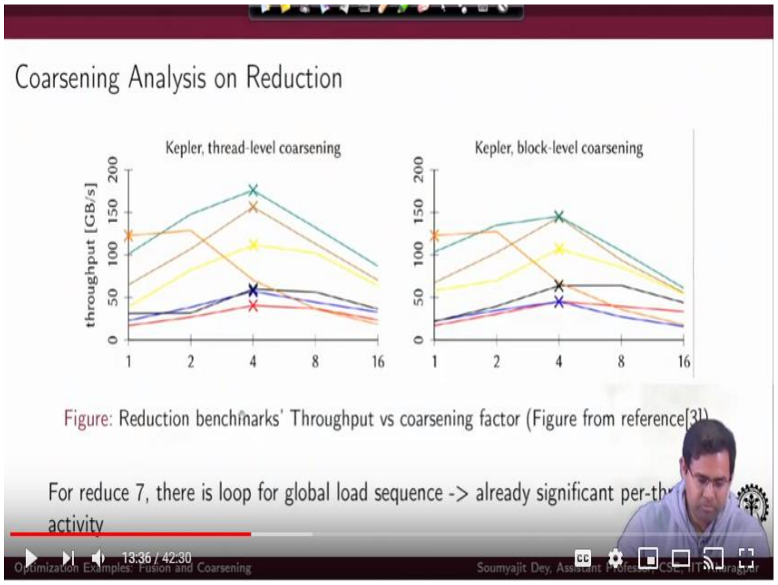
So this is the step from where you start increasing the part thread activity. So this is like some example of coarsening inside the reduction kernels itself. So you use actually n/2 number of threads then in reduction 5 instruction 5 you unrolled the last warp and you were able to remove intra warp synchronization barriers for the last warp and all that.

And if you remember in reduction 6 you performed a template parameter based complete unrolling of the kernel based on whatever is the block size you will be generating code where there was effectively no loop and it was a completely unrolled version of the kernel right. So all the loop overheads were removed and then in reduction 7 you had multiple elements per thread and you launched very small constant number of thread blocks.

So that was also somewhere you have increased the part shared activity significantly and it was requiring very few synchronization barriers right. So the point we are trying to make here first of all these are the reduction steps we are referencing some of these we have discussed up to some level of detail and you can easily find the further detail of all these reduction steps in the Nvidia example of reduction kernel optimization which is easily available online and which has been referenced in our earlier lectures.

So the point we are trying to make here is with each higher step of production we are increasing the part thread activity significantly and actually removing lot of redundancy and unnecessary barriers and unnecessary loop overheads and things like that.

(Refer Slide Time: 13:36)



Coarsening Analysis on Reduction

Kepler, thread-level coarsening
Kepler, block-level coarsening
throughput [GB/s]
0
50
100
150
200
1
2
4
8
16

Figure: Reduction benchmarks' Throughput vs coarsening factor (Figure from reference[3])

For reduce 7, there is loop for global load sequence -> already significant per-th activity

13:36 / 42:30
Optimization Examples: Fusion and Coarsening
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So the point here is that if we are actually applying thread coarsening into this different variants of these reductions we see that how things perform right. So here we do an analysis of how coarsening effects the execution time for each of these reduction kernels okay. So in this plot we provide the examples of this reduction kernels.

So here what we have is this different reduction kernels and their overheads in terms their peak throughputs and how the throughputs keep on changing with the different coarsening factors. So this is the plot for reduction 7 and the other plots are for the reduction kernel this is for the reduction kernel 1 then the 2, 3, 4 ,5 ,6 and this is the 7th kernel so they are in this order sorry the legend in the figure is missing here.

So I hope I mean you can see that with more amount of optimization the initial speed up as you can see in terms of throughput there is it keeps on increasing right. So this reduction 1 this reduction 2 3 4 5 6 and 7 right. Now observe firstly that and so here we are using a Kepler architecture based system tastes like a 40 curve in our system and we are applying a thread level coarse we are actually using 2 different kinds of optimizations.

One optimization is the I mean coarsening optimization we are applying thread level coarsening and we are applying block level coarsening. So the first important characteristics we see here is that the optimal coarsening factor for all of the first 6 reduction kernels appears to be the coarsening factor of 4 both for thread level and block level coarsening.

This is for our system and for this specific example of reduction and for reduction kernel 7 we see that coarsening does not really help it first remains constant or increases a bit in terms of the peak throughput right. So that by the way that is the index its telling you that how many how I mean that what is the throughput of the kernel and with coarsening is going to increase.

But however for reduction 7 we see that its gradually decreasing right by the way we like to say that this is a plot where reproduce I mean we are actually reproducing from this reference which is actually highlighted here at the end. So this is from this paper and these results actually are given from this paper right we also have this kernel versions but these specific plots that we have given they have been directly taken from this paper I mean you can also try them we have tried these programs and the I mean it is very easy to reproduce this behaviour in your own system.

Of course the coarsening factors may be it may be different but this is an example and the plots which you have directly getting from this reference here right. So what they are trying to show is how far the reduction kernels the activities keep on changing and as I was saying that for reduction 7 you see that there is no real effect of coarsening but for all the others it seems that coarsening by a factor of 4 has 5 is actually providing you the peak throughput.

So this is the observation first brought out by the authors of this paper which appeared recently now if we try to analyse we thought that this would be a very nice example for this specific part course material on thread coarsening because it really brings out the effect that coarsening has in terms of the occupancy of the architecture okay first of all let us understand why for reduction 7 there is no significant speed up.

Now if you remember what was happening in reduction 7 in reduction 6 so first of all we started the coarsening of threads in terms of increasing not really coarsening of threads but actually increasing part thread activity from reduction 4 right. So I would not say we coarsened threads by we just simply delegated some activities for the threads it was not in the semantics of coarsening and so we will not say that for.

If you see from reduce 4 we delegated some activity to the threads right it was in terms of performing more number of global loads right. But then it was only normal part thread activities so we should not say it is not coarsening but just increasing the part thread activity but that also increases the amount of resources the threads are going to consume in terms of memory traffic here in terms of storage in the registers here.

Then it reduce 4 it was simple you just unrolled the last warp for reduction in reduce 6 we completely unrolled so these are more or less 5 and 6 or optimizations where you remove the loop overheads and you remove the synchronization over it specifically introduced 5 right reduced experiment the loop overheads but what happens in reduced 7. In reduced 7 the idea of making each thread warps some more is being generalized.

So instead of making each thread bring some only 2 data points we can make each thread bring multiple more number of data points and make them do the local additions on global memory data. So each thread is given more amount of sequential activity in general. So let us say they are adding up they are bringing in data elements from the global memory adding them up and then starts the usual reduction behaviour right.

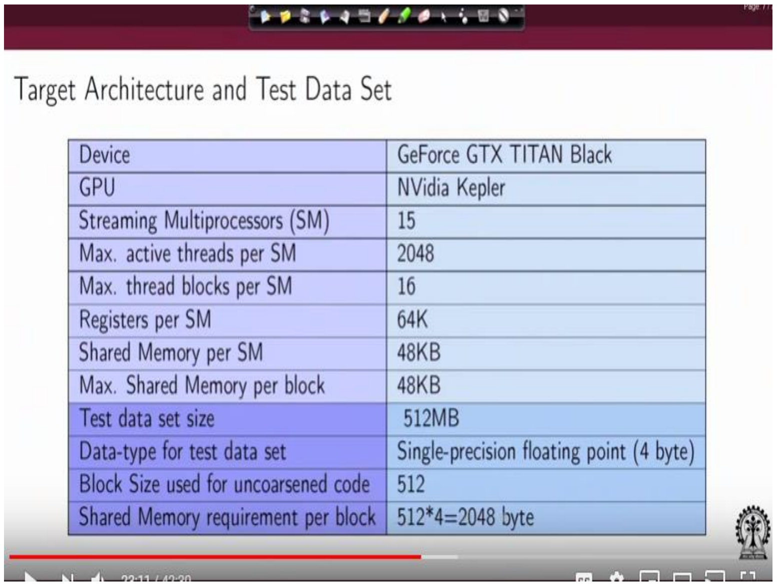
So reduce 7 actually is a good optimization where you can select the amount of sequential addition each of the threads are going to do such that overall we have the perfect occupancy in terms of the reduction kernel right. So as you can see with each reduction step we are doing an optimization at some point we have reached all the possible optimizations. Now what you are trying to do is you are trying to increase the part thread activity so that you get the perfect amount of occupancy right.

So the perfect amount of occupancy can be achieved if whatever is the total number of threads you are able to schedule in your GPUs SM you make all the threads there is this peak number of threads used all the SM resources maximally right. So then that is your perfect notion of occupancy and which it seems that with a suitable amount of with suitable choice of part thread activity in terms of number of sequential loads.

I can tune this value and make reducing 7 achieve that perfect occupancy. Now once we do that I hope it is clear to us that once we are able to tune that perfect occupancy just by increasing the sequential activity per thread with respect to reduced 7 further coarsening is not really going to help because whatever is the peak parallelism to be explored is already done with reduced 7 right.

So that that explains this figure which has been reported by the this publication reference and but let us now try and analyse in our terms that why really these authors were able to get a peak tribute in the Kepler architecture for the other reduction kernels at a coarsening factor of 4.

(Refer Slide Time: 21:26)



Target Architecture and Test Data Set

Device	GeForce GTX TITAN Black
GPU	NVidia Kepler
Streaming Multiprocessors (SM)	15
Max. active threads per SM	2048
Max. thread blocks per SM	16
Registers per SM	64K
Shared Memory per SM	48KB
Max. Shared Memory per block	48KB
Test data set size	512MB
Data-type for test data set	Single-precision floating point (4 byte)
Block Size used for uncoarsened code	512
Shared Memory requirement per block	512*4=2048 byte

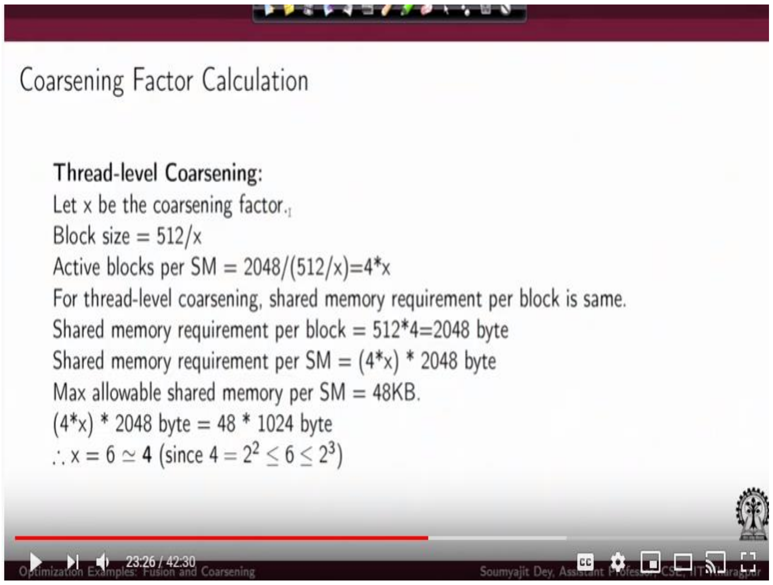
So if you look into the target architecture and the test data set for this warp. So this is some analysis we are doing and maybe this is a very good example where we are really talking about optimization in terms of not only algorithmic optimization but also the optimization with respect to the architectural parameters that are there so here the device under test is a Nvidia GeForce GTX TITAN Black GPU for this specific warps by the authors and their GPU is a Kepler architecture number of SMs is 15.

The SMs allow maximum active threads per SM as 2048 maximum thread blocks per SM as 16. So observed these are very important parameters the maximum number of active threads allowed is 2048 maximum thread blocks allowed is 16 the number of registers in the register file inside this SM is 64 K that is the size file size and then shared memory is 48 KB and the maximum shared memory provide allowable per block is also the same.

So right and the test data size the data for which the reduction has been performed is a 512 megabytes right. The data type is single precision floating point in this for the reduction kernel so that is 4 byte. So if we consider the block size for the original uncoarsened code it has been taken as 512 and so since the block so I have 512 for data points and each of them are floating point type.

So the shared memory requirement for each block is also to 2048 bytes right. So this is important in the uncoarsened version the shared memory requirement power block is 2048 total amount of shared memory available in the SM is 48 kilobytes right.

(Refer Slide Time: 23:20)



Coarsening Factor Calculation

Thread-level Coarsening:

Let x be the coarsening factor.

Block size = 512/x

Active blocks per SM = 2048/(512/x)=4*x

For thread-level coarsening, shared memory requirement per block is same.

Shared memory requirement per block = 512*4=2048 byte

Shared memory requirement per SM = (4*x) * 2048 byte

Max allowable shared memory per SM = 48KB.

(4*x) * 2048 byte = 48 * 1024 byte

$\therefore x = 6 \simeq 4$ (since $4 = 2^2 \leq 6 \leq 2^3$)

23:26 / 42:30
Optimization Examples: Fusion and Coarsening
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So let us figure out what is a good coarsening factor here let x be the coarsening factor right so what is the block size we are saying that effectively the block size if I am doing thread level coarsening so I am going to reduce the threads per block so that would be reduced by original block size of 512.

So this was the original block size 512 that would get reduced by x right 512/x is the reduced block size with this what would be the number of active blocks per SM. So as you can see that you have maximum active threads per SM is 2048 right. Now there is a maximum allowed limit on active threads per SM. So now it is going to be active blocks per SM would be original total 2048 threads and threads per block is 512/x. So that would give me 2048 divided by this.

So that gives me 4x right of course the other limiting factor is the maximum thread blocks per SM which is 16 we are just hoping that we will be reaching the threshold by this limiting factor otherwise you have to consider the other limiting factor also right. So for the thread level coarsening the shared memory requirement per block remains same as we know because we are we are coarsening the threads and reducing the threads per block simultaneously.

So per block remain is same so what is the requirement so shared memory requirement per block is 512 times 4 which is 2048 byte as we have required computed earlier already and in that case since we know that active blocks per SM would be 4x. So what is the shared memory requirement overall part is same so that is 4x times 2048 byte and of course the maximum allowable shared memory in the SM is 48 KB.

So here we have an equation so we have 48 times 1024 and that should be equal to this 4x times 2048 byte right. So 48 KB that is multiplied by 1024. So already you get x=6 now if we put the nearest value in terms of I mean here if you just get the lower value in terms of that powers of 2 so you get it as 4 since you have it the coarse you can use so since its 6 so you have to go to the next lower power of 2 will be coarsening always in powers of 2 like that and that would give you the 4 here right.

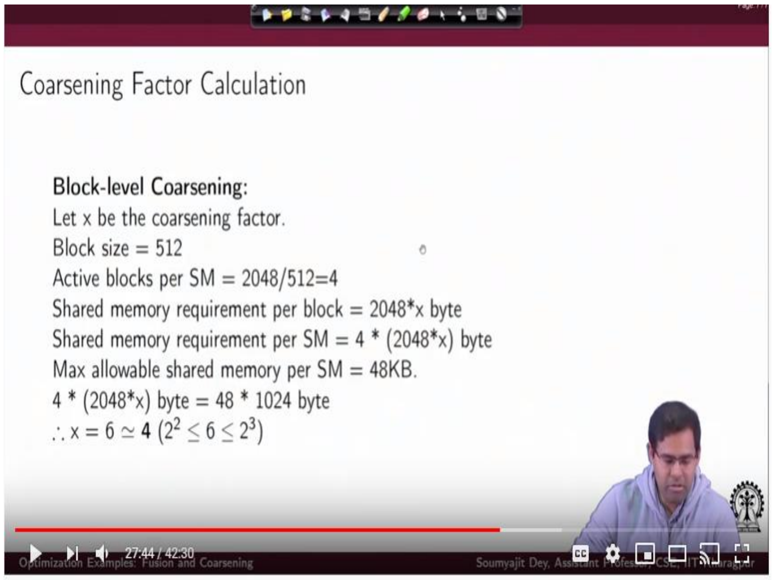
So with that we see that if we choose 4 we get the part of coarsening factor as 4 here out of this calculation that with this value of 4 we are making a use of shared memory such that we have the maximum possible occupancy. If we go beyond this then what would happen is that with further coarsened threads the number of blocks that you can read the number of threads that you can really schedule they will go to be less than the number of active threads per SM.

Because they will keep on requiring more amount of shared memory right. So this is a possible calculation that you can do for thread level coarsening but of course you can here in this calculation we are just using the limiting factor which is the number of active blocks per SM where the limiting factor is taken as 2048 you can just check whether the other limiting factor which is threads per block per SM is also limiting this value of x is just a take home you can just check it and see whether with that if I consider that constant.

I hope you are understanding that here we are considering the constant of the maximum number of active threads and computing using that okay in that case what can be the number of active blocks right but we also know that with this the maximum number of thread blocks is 16. So here with x=4 I get this as 16 right. So that anyway would mean that I cannot go beyond this because with a higher factor what would happen is it may happen that you can get you can also hit this limiting factor.

So in general you have to take care of both you can do a calculation and figure out whether if you just start with this as the limiting factor what value do you get and whether it hits the other barrier in terms of the other limiting factor which is this right they are kind of related.

(Refer Slide Time: 27:44)



Coarsening Factor Calculation

Block-level Coarsening:
Let x be the coarsening factor.
Block size = 512
Active blocks per SM = 2048/512=4
Shared memory requirement per block = 2048*x byte
Shared memory requirement per SM = 4 * (2048*x) byte
Max allowable shared memory per SM = 48KB.
4 * (2048*x) byte = 48 * 1024 byte
$\therefore x = 6 \simeq \mathbf{4}\ (2^2 \leq 6 \leq 2^3)$

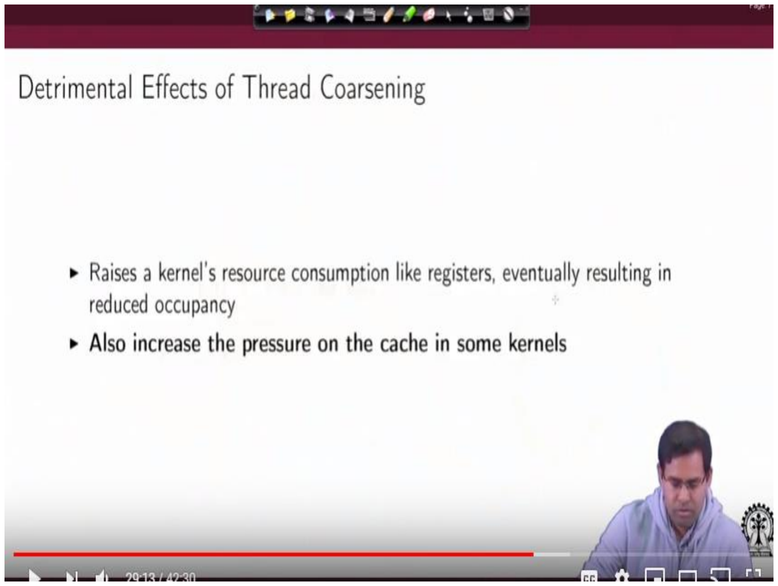
27:44 / 42:30
Optimization Examples: Fusion and Coarsening
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So you can just replicate a similar analysis for a block level coarsening. So choose x as a coarsening factor block size is known for the original uncoarsened version of the kernel. Active blocks for SM would be in that case for like us like the original and so shared memory requirement per block you can just.

So here will be what it will be it will increase by the x value right shared memory requirement per block whatever was original it multiplies by the x value and then shared memory requirement per SM would be multiplied by the requirement per block multiplied by the number of blocks. So you multiply by 4 so then you can just have a equation where you have this shared memory required per SM equal to the maximum allowed shared memory which is 48 KB.

So 48 multiplied by 1024 and you get the similar calculations again right. So these are the possible ways in which you can also empirically derive limit on the coarsening factor. But of course there are other issues in the architecture in terms of nonlinear relationships between the different factors like interference and other issues we are not getting to that detail. We are just trying to provide a handle here like how you can analyse that why possibly you are reaching this kind of a peak performance with this coarsening factor here.

(Refer Slide Time: 29:13)



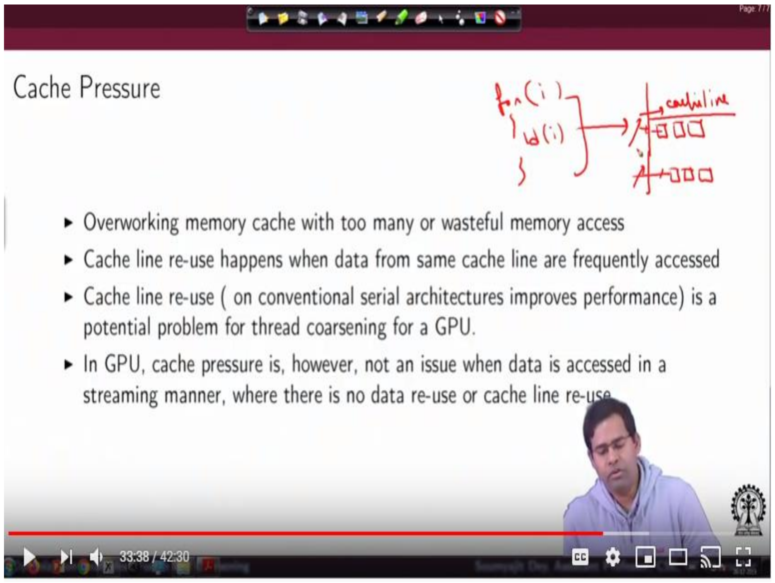
Detrimental Effects of Thread Coarsening

- Raises a kernel's resource consumption like registers, eventually resulting in reduced occupancy
- Also increase the pressure on the cache in some kernels

So there are several other effects of detrimental effects of thread coarsening which also gets exemplified with this picture. As you can see that the peak throughput we are getting with the coarsening factor 4 for both for thread and block level but then the peak throughput reduces for increased coarsening factors of 8 and 16.

Now we have discussed this that with higher amount of coarsening I can have reduced occupancy right and the reason being the kernels resource consumption I mean increases right parts read in part thread resource consumption in terms of registers and share memory increases and that is resulting in the reduced occupancy. The other important thing is this also creates an increase in the pressure on the kernel.

(Refer Slide Time: 30:07)



Cache Pressure

- Overworking memory cache with too many or wasteful memory access
- Cache line re-use happens when data from same cache line are frequently accessed
- Cache line re-use (on conventional serial architectures improves performance) is a potential problem for thread coarsening for a GPU.
- In GPU, cache pressure is, however, not an issue when data is accessed in a streaming manner, where there is no data re-use or cache line re-use

So this is an important parameter which is cache pressure right so of a few words on cache pressure. So if you are overworking the memory cache with too many or wasteful memory accesses then it is a bad thing and that is the effect on the cache is used I mean modelled by this term cache pressure right.

Now it may happen that you are reusing a cached line time and again by a single thread. So which is which is not a good idea for a GPU so and that can create issues. So when I am doing cache line reuse where for data that means 1 thread is accessing the same cache line multiple times its considered bad for the GPU.

Because in the GPU I would like to have consecutive thread ideas inside a warps accessing sequential data elements and that would make that would mean a single warp makes a single cache line access and brings in the data rather than 1 thread bringing data multiple times in different iterations of a loop or similar kind of activities.

So whenever there is cache line reuse there is a bad thing in a GPU unlike a conventional serial architecture where its considered a good thing right. So if you have a GPU kernel where a thread is accessing a cache line multiple times. Let us say you have a for loop using which the thread is accessing consecutive data elements then that is a bad thing ideally you should not be writing the code like that.

But if you have if the programs nature is such that the thread is accessing consecutive locations then in terms of warps is going to access the cache line multiple times by the same thread which is good in the sequential or serial architecture but not in the GPU. Now consider that you are cautioning this kind of a program.

So then this thread will be performing its own execution of a for loop where its making multiple accesses of a cache line and it will also be doing the activity of another originals or another thread in the original kernel and for that also it will be sequentially accessing the cache that the elements inside the same cache line.

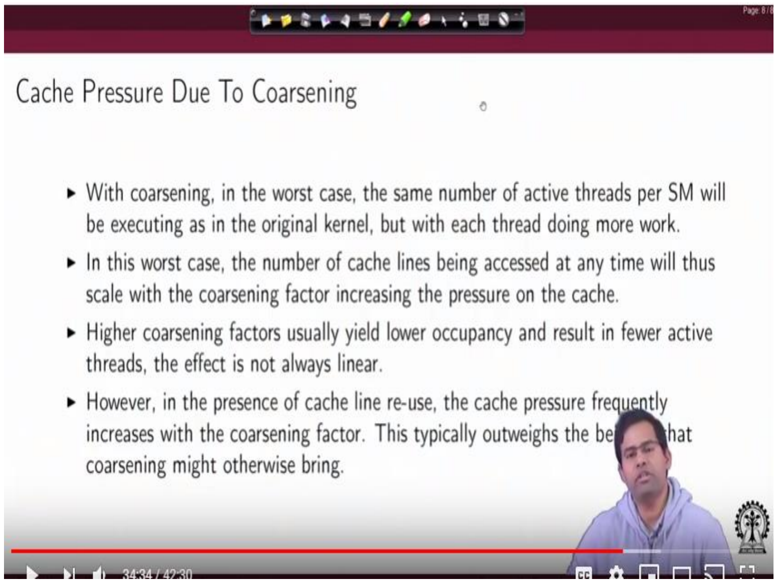
So if the original kernel has got lot of instances of part thread cache line reuse in the coarsened kernel there will be further iterations. So just if I am trying to take a picture here that suppose you have a for loop through which 1 thread a for a you have an active a behaviour says that 1 thread id is accessing the same cache line for consecutive memory locations right.

So in so there is one load instruction here and it is increasing with the index i and the for loop is iterating over i and 1 thread is looking into the same cache line time and again and it is making a cache line reuse for this kind of a code when I do the coarsening. So then the thread will do this behaviour for multiple cache lines right because it will also replicate some other threads activity and again do cache line reuse well it was originally doing cache line reuse only in one place.

So this is bad thing in this for a first of all this is a bad programming behaviour for a GPU ideally this sequential access should have been delegated to sequential threads but if that the nature of the algorithm as I am saying. So then you are increasing cache line reuse by a coarsening factor which is significantly increasing the cache pressure and due to this the overall kernels throughput will decrease right.

So there is the impact of cache pressure on a program and the coarsening will hamper it further right.

(Refer Slide Time: 34:10)



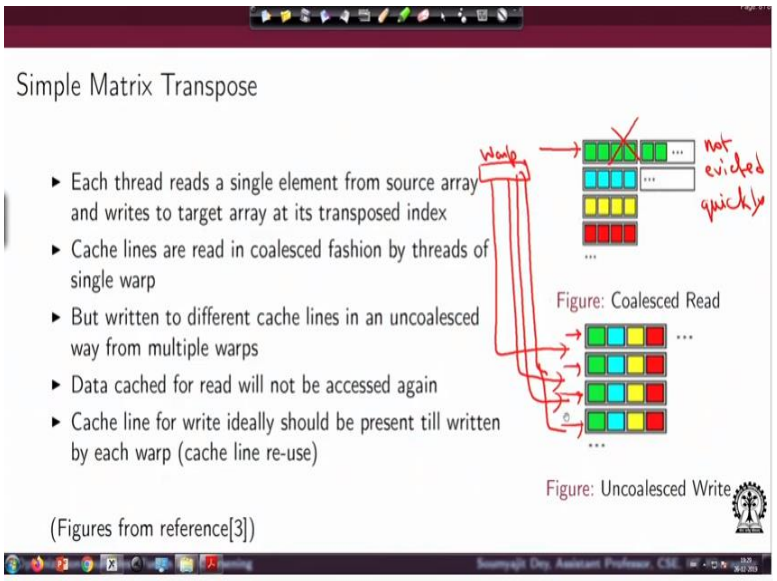
Cache Pressure Due To Coarsening

- With coarsening, in the worst case, the same number of active threads per SM will be executing as in the original kernel, but with each thread doing more work.
- In this worst case, the number of cache lines being accessed at any time will thus scale with the coarsening factor increasing the pressure on the cache.
- Higher coarsening factors usually yield lower occupancy and result in fewer active threads, the effect is not always linear.
- However, in the presence of cache line re-use, the cache pressure frequently increases with the coarsening factor. This typically outweighs the be hat coarsening might otherwise bring.

So in general I can say that if I do a coarsening in the worst case the same number of active threads per SM will be executing as in the original kernel. But now each thread will of course be doing more warps and due to its each thread doing more work. If the work is to load the cache line multiple times then the number of cache lines accessed will also scale win the coarsening factor just as we are saying and with higher amount of coarsening factors will have more higher cache line reuse.

And that will effectively I mean walk down the SM and decrease the occupancy in that way and this will significantly outweigh the benefit of performing any kind of coarsening whatsoever right.

(Refer Slide Time: 34:57)



Simple Matrix Transpose

- Each thread reads a single element from source array and writes to target array at its transposed index
- Cache lines are read in coalesced fashion by threads of single warp
- But written to different cache lines in an uncoalesced way from multiple warps
- Data cached for read will not be accessed again
- Cache line for write ideally should be present till written by each warp (cache line re-use)

(Figures from reference[3])

Now let us take another example of the matrix transpose kernel which we know as a its a quite difficult kernel and there are lot of optimizations which we have already discussed. Now let us have a relook into this from the point of view of coarsening. So in the single matrix transpose kernel each thread will read a single element from the source array and write to the target array let us consider the situation where we are doing coalesced reads whereas we follow un coalesced write.

SO the cache lines are read in coalesced fashion by threads in the single warp because we are doing coalesced read. So there is no much of cache line reuse in terms of read but when the writes are going to happen they are uncoalesced. So writing to the different cache lines is happening in an uncoalesced way in multiple different warps right.

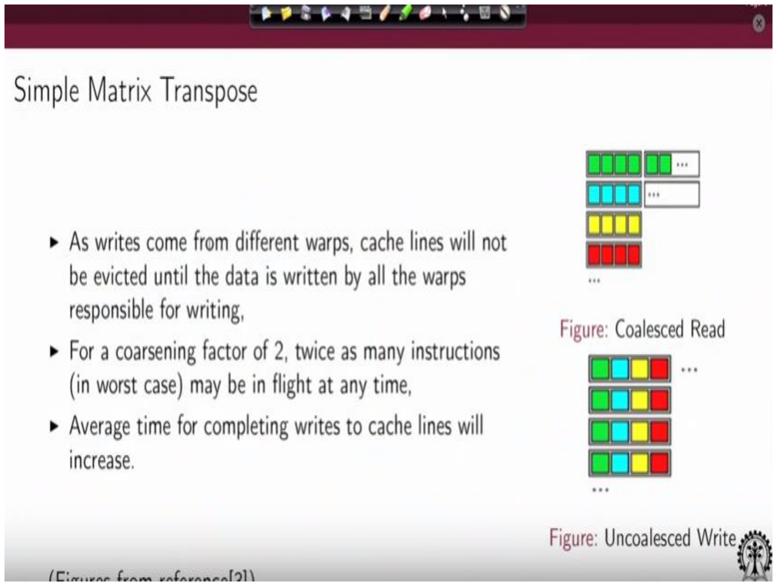
So data cached for a read first of all is not going to be accessed again. So there is the thing that when you are reading from here you are reading in one shot right so you are reading in one shot after the data has been read this cache line is not going to be accessed. But that does not mean immediately after the read has happened the cache line will be evicted. Why? because the write is the read is performed quickly.

But then the write is going to happen column wise in a uncoalesced way from multiple warps. So that the right is going to happen in different in multiple cycles right. So for the same warp so let us say this is the warps which has performed the read this warp is going to load the data here in the right in multiple cache transactions right so there will be multiple cache transactions happening.

So at any point of time we should be saying that well there are lot of write instructions in flight since there are a lot of write instructions in flight and the average latency of the cache flight is very high and that effectively increases the amount of time for which I am unable to evict the data from a cache line.

I hope this is clear as long as I am not able to read as well as write well the data column wise I cannot evict data from the cache right. Now since the writes are getting serialized in this case I am unable to perform you know I am unable to actually evict the data. So that creates significant delay and decreases at throughput.

(Refer Slide Time: 37:52)



Simple Matrix Transpose

- As writes come from different warps, cache lines will not be evicted until the data is written by all the warps responsible for writing,
- For a coarsening factor of 2, twice as many instructions (in worst case) may be in flight at any time,
- Average time for completing writes to cache lines will increase.

Figure: Coalesced Read

Figure: Uncoalesced Write

(Figures from reference[2])

So as there again as the writes are coming from the different warps the cache line will not be evicted until the data is written by all the warps responsible for the writing. So the read is coalesced write is un coalesced write takes multiple cache cycles. So I cannot evict the data from the cache even immediately after the read happens right. So for a coarsening factor of 2 I will have twice as many instructions which may be in flight at any time.

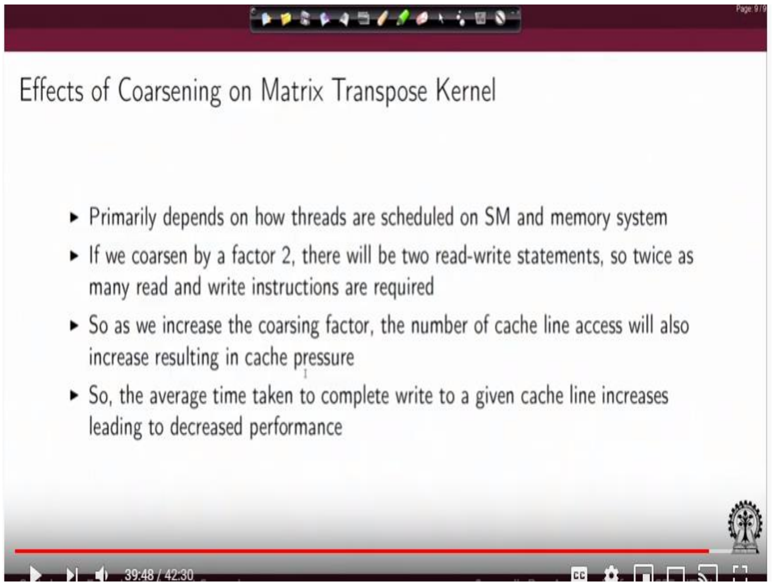
So I hope you are understanding that if we consider this transpose operation already there was this difficulty with the write and there was this issue with the cache right now if I increase the coarsening factor earlier whatever was the number of instructions which were in flight at any time that would increase by the factor itself right.

So the average time for completing the rights to the cache are going to increase it was already high the average time was already high the average time for the cache line eviction was high and since the right time will increase further due to the coarsening. The coarsening increases the part thread activity the coarsening will increase the amount of data to be written to the column.

So we will just go back again if we coarsen what happens now we will be reading more amount of data in the coalesced read. So again we will be writing more amount of data in the coalesced write by inside the single thread I am not having the advantage of having multiple threads to do the writes right. So the amount of sequential column wise access here for the data will increase.

So that would further increase the cache line reuse here I mean that would further increase the amount of time for which I cannot evict the data from the cache right.

(Refer Slide Time: 39:31)



Effects of Coarsening on Matrix Transpose Kernel

- Primarily depends on how threads are scheduled on SM and memory system
- If we coarsen by a factor 2, there will be two read-write statements, so twice as many read and write instructions are required
- So as we increase the coarsing factor, the number of cache line access will also increase resulting in cache pressure
- So, the average time taken to complete write to a given cache line increases leading to decreased performance

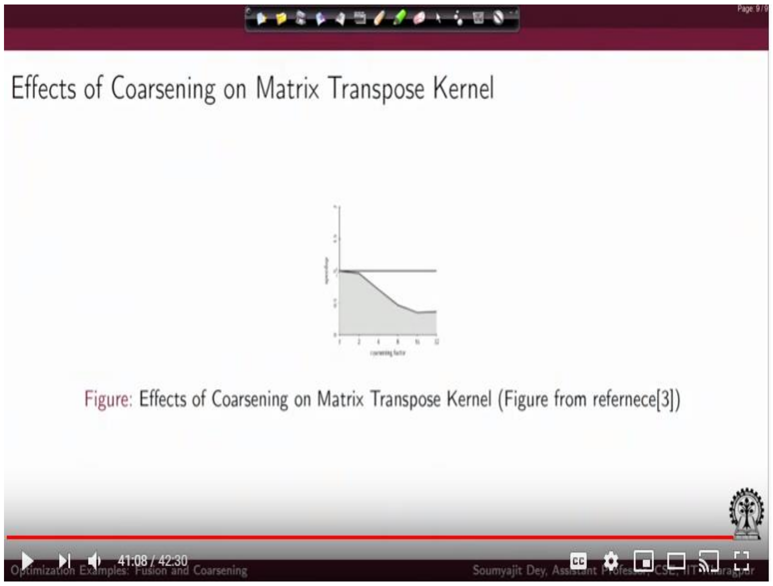
So this also will create a problem here and so just as an example this was our uncoarsened version of the transpose kernel we have seen earlier. So you just read you compute the index in row wise input and you write column wise right and for this kind of a non-optimized matrix transpose kernel the effect of coarsening is severely bad right.

So it primarily depends on how threads are scheduled on the SM and memory system. Therefore there is a dependency and if we questioned by a factor of 2 as we are saying there will be 2 read write statements so twice as many read and write instructions are required here twice as many of course. Because I am increasing the part thread activity by 2 and as we increasing the coarsening factor the number of cache line access will also increase right.

Because now each thread will access more and now that would effectively increase the cache pressure and therefore as time taken to complete the write to the cache line increases as we have discussed since that time taken to complete the write increases it also increases the overall latency and it decreases the performance in terms of as we have seen from the picture that after I mean for the reduction kernel there was a decrease.

But for transpose doing any kind of coarsening is going to be difficult specially for this transpose kernel where we have this kind of a column wise write and row wise reads right.

(Refer Slide Time: 41:01)



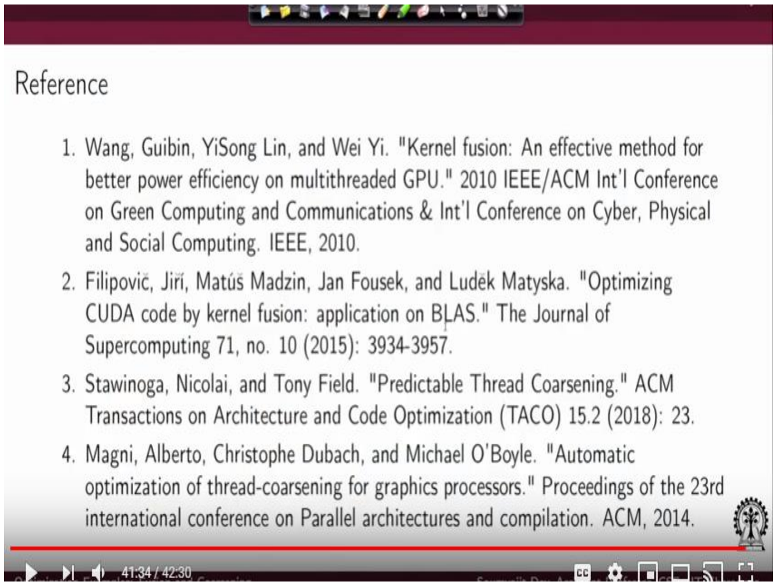
Effects of Coarsening on Matrix Transpose Kernel

Figure: Effects of Coarsening on Matrix Transpose Kernel (Figure from refernece[3])

41:08 / 42:30
Optimization Examples: Fusion and Coarsening
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So this is the simple example here this is a picture again taken from the reference paper we are talking about that for this kind of a matrix transpose by the way this is the optimized matrix transpose that we discussed earlier in our lectures on matrix transpose as you can see any kind of coarsening is actually creating the cache coarsening and cache reissue and create increasing the latency of the writes and due to that any kind of coarsening is linked to a hamper in the leading to complete hampering of the speed-up right.

(Refer Slide Time: 41:34)



Reference

1. Wang, Guibin, YiSong Lin, and Wei Yi. "Kernel fusion: An effective method for better power efficiency on multithreaded GPU." 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing. IEEE, 2010.
2. Filipovič, Jiří, Matúš Madzin, Jan Fousek, and Luděk Matyska. "Optimizing CUDA code by kernel fusion: application on BLAS." The Journal of Supercomputing 71, no. 10 (2015): 3934-3957.
3. Stawinoga, Nicolai, and Tony Field. "Predictable Thread Coarsening." ACM Transactions on Architecture and Code Optimization (TACO) 15.2 (2018): 23.
4. Magni, Alberto, Christophe Dubach, and Michael O'Boyle. "Automatic optimization of thread-coarsening for graphics processors." Proceedings of the 23rd international conference on Parallel architectures and compilation. ACM, 2014.

So with this we like to end our discussions on the different on thread coarsening is good and bad ways I hope we have performed a good architectural level analysis specifically for these examples. So you can really think more in these terms for other examples and the other example algorithms and their course learnings and here are some of the references important references which we used for this lecture material.

These are reference on kernel fusion which we extensively used then the second reference on kernel fusion. These are all some of them are quite recent papers actually and the thread coarsening part was almost entirely taken from this predictable thread coarsening paper which appeared very recently and of course this is another paper on optimizations with respect to thread coarsening right. So with this we will be ending this long lecture and thank you for your attention.