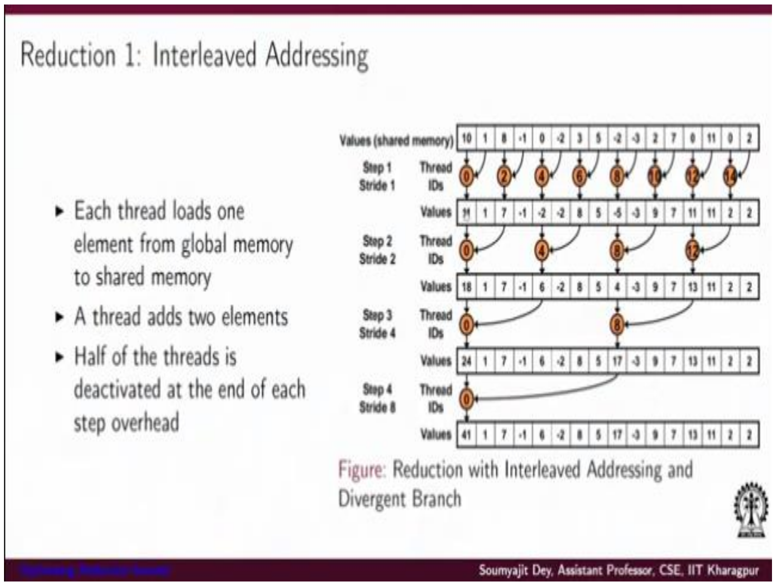
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology - Kharagpur

Lecture – 29
Optimising Reduction Kernels (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming so, just as a small recall, in the last lecture we have been discussing of I mean, about 3 I mean, possible optimisation techniques that you can while writing a code for parallel reduction in a GPU architecture.

(Refer Slide Time: 00:45)



Reduction 1: Interleaved Addressing

- Each thread loads one element from global memory to shared memory
- A thread adds two elements
- Half of the threads is deactivated at the end of each step overhead

Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1 Stride 1
Thread IDs 0 2 4 6 8 10 12 14
Values 11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
Step 2 Stride 2
Thread IDs 0 4 8 12
Values 18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
Step 3 Stride 4
Thread IDs 0 8
Values 24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
Step 4 Stride 8
Thread IDs 0
Values 41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

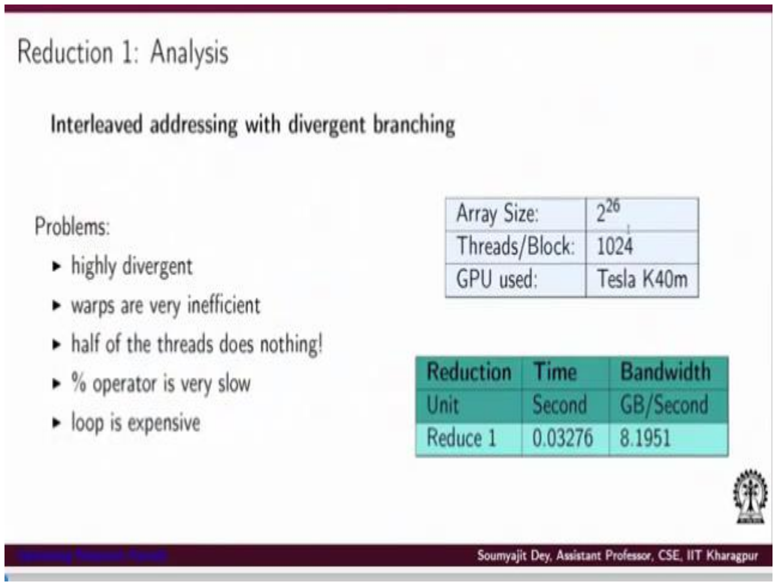
Figure: Reduction with Interleaved Addressing and Divergent Branch

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, we just picked up the problem and discussed a few optimisation techniques, so a small recap on the techniques we already discussed so, the first was the very nice parallel algorithm, so where we just used threads for doing the local computation. So, the first thing is that you just use the threads to load the data from the global to the shared memory and then each thread is doing a sum of consecutive elements and in that way, what you are essentially doing is you are using thread Ids with alternate values.

I mean, their thread Id is differing by 1 here right, to add consecutive values and you go to the next iteration, so what is happening is in each iteration, I mean half of the threads are going idle and also, in parallel the threads are doing the addition in strides which are increasing by a factor of 2, right.

(Refer Slide Time: 01:52)



Reduction 1: Analysis

Interleaved addressing with divergent branching

Problems:

- highly divergent
- warps are very inefficient
- half of the threads does nothing!
- % operator is very slow
- loop is expensive

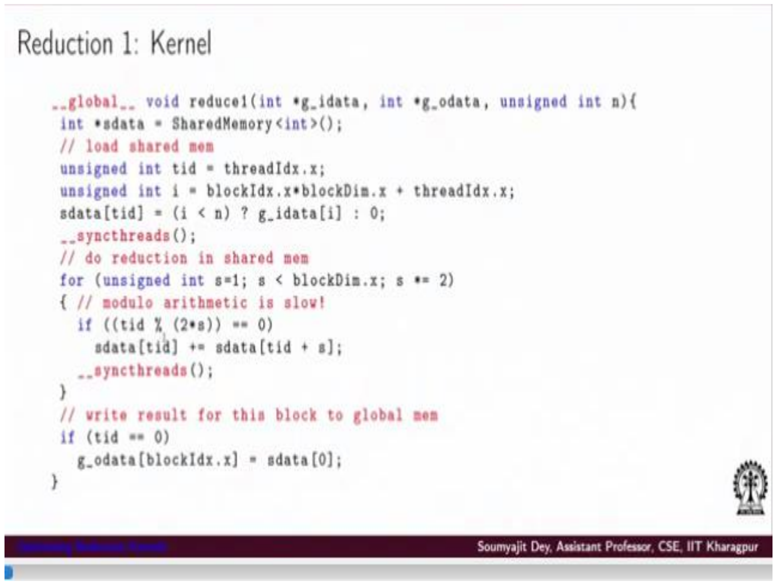
Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And the problems, that we had with this implementation was as follows like; first of all, it was a highly divergent implementation.

(Refer Slide Time: 01:57)



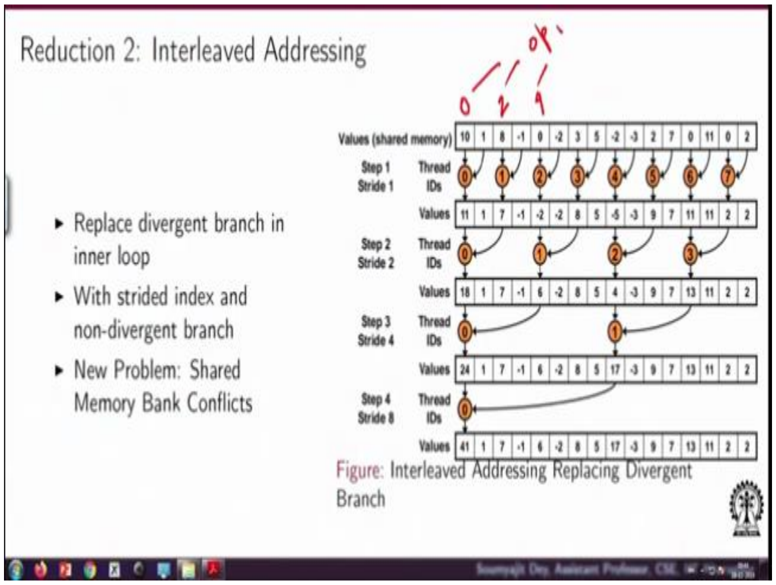
Reduction 1: Kernel

```
__global__ void reduce1(int *g_idata, int *g_odata, unsigned int n){
 int *sdata = SharedMemory<int>();
 // load shared mem
 unsigned int tid = threadIdx.x;
 unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
 sdata[tid] = (i < n) ? g_idata[i] : 0;
 __syncthreads();
 // do reduction in shared mem
 for (unsigned int s=1; s < blockDim.x; s *= 2)
 { // modulo arithmetic is slow!
   if ((tid % (2*s)) == 0)
     sdata[tid] += sdata[tid + s];
   __syncthreads();
 }
 // write result for this block to global mem
 if (tid == 0)
   g_odata[blockIdx.x] = sdata[0];
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

As you can see due to this if condition, half of your threads inside the same warp is getting; I mean is getting to diverge here at this point and also and you have a loop over it and you are doing percentile computation that also has got some over it, right.

(Refer Slide Time: 02:27)



0 2 4
op.
Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1 Stride 1
Thread IDs 0 1 2 3 4 5 6 7
Values 11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
Step 2 Stride 2
Thread IDs 0 1 2 3
Values 18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
Step 3 Stride 4
Thread IDs 0 1
Values 24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
Step 4 Stride 8
Thread IDs 0
Values 41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

Reduction 2: Interleaved Addressing

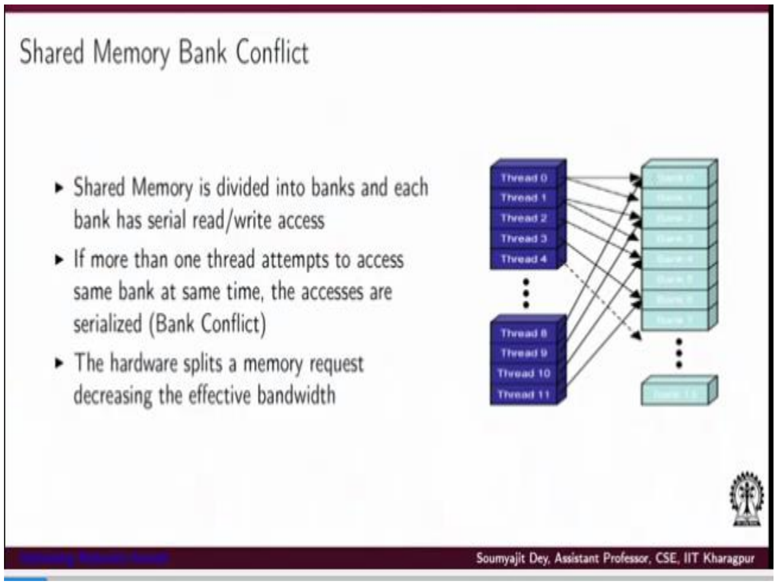
- Replace divergent branch in inner loop
- With strided index and non-divergent branch
- New Problem: Shared Memory Bank Conflicts

Figure: Interleaved Addressing Replacing Divergent Branch

And so, these were the most important problems here. The first thing we do as part of optimisation 2; in this case, the reduction 2, I would say the first optimisation here is that you make consecutive threads to the addition. So, if you remember the earlier picture, the thread Ids were 0, 2, 4, like that but we do some program transformation and make consecutive threads take care of doing the addition.

So, we have the other features constant that means the strides keep on increasing in multiples of 2 but you are working with consecutive threads. Due to this, you always have warps which are non-divergent because all consecutive threads are always active most of the time but the problem here is as we can see that you have a situation of shared memory bank conflict which we can look into by studying the access pattern of the threads.

(Refer Slide Time: 03:25)



Shared Memory Bank Conflict

- Shared Memory is divided into banks and each bank has serial read/write access
- If more than one thread attempts to access same bank at same time, the accesses are serialized (Bank Conflict)
- The hardware splits a memory request decreasing the effective bandwidth

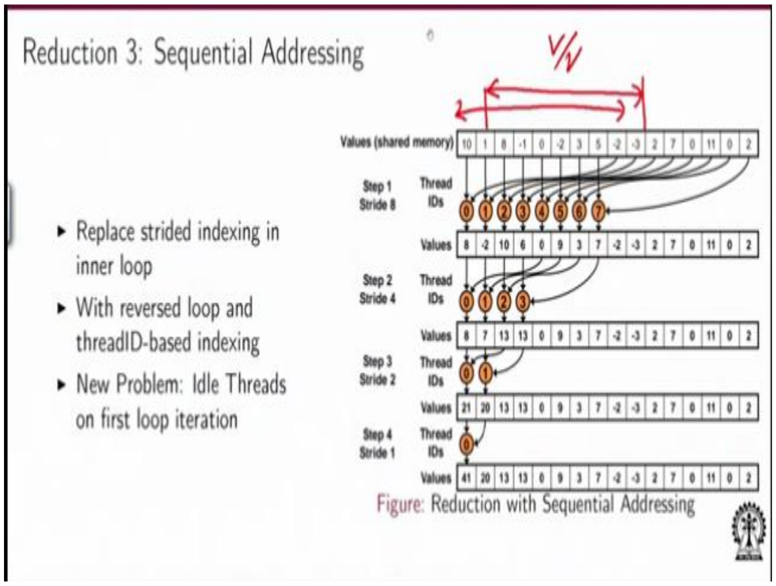
So, every thread is accessing 2 consecutive values and this will give rise to a 2 way bank conflict for each thread here, in this example the solution here would be that okay, let us now change the way in which the threads access the corresponding values. So, if you just take a close look here; here, we made consecutive threads warp but consecutive threads are working on 2 consecutive values from the shared memory, right.

So, when the operation is going to happen, for each thread first there would be a load, right, I mean for operand 1, load for operand 2 and then the addition, right. Now, here since, if I look at the access pattern for the threads, all the threads, that the loads that we are going to do for operand 1, they are not consecutive, that is the basic problem, right. So, thread 0 is loading from here thread 1 is loading from here, thread 2 is loading from here as you can see.

So, to be more specific so, thread 0 is loading from location 0, thread 1 is loading from location 2, thread 2 is loading form the location 4 and all that for operand 1, right, so that is the instruction that will fire and when this instruction fires; all the threads in the warp, since they are not accessing exactly consecutive locations, so we have got this issue of the bank conflict here but how do we remove that?

Now, let us look at the access pattern of a transform thread for a transformed operation here, so let us look at the alternate code that we can have to resolve the bank conflict.

(Refer Slide Time: 05:23)



Reduction 3: Sequential Addressing

- Replace strided indexing in inner loop
- With reversed loop and threadID-based indexing
- New Problem: Idle Threads on first loop iteration

V/N
2
Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1 Stride 8
Thread IDs 0 1 2 3 4 5 6 7
Values 8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 2 Stride 4
Thread IDs 0 1 2 3
Values 8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 3 Stride 2
Thread IDs 0 1
Values 21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 4 Stride 1
Thread IDs 0
Values 41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Figure: Reduction with Sequential Addressing

So, now let us start saying that okay, let us arrange the threads in such a way that yes, I have consecutive threats working, so that is removing the issue of divergent inside the warp, also we change the way the consecutive thread Ids access the memory. So, what we do is; earlier thread Id is 0, was accessing memory location 0 and memory location 1 but now, let us make thread Id 0, access memory location 0.

And the next element that is going to access is sitting at an offset which is of size L by 2, right, where L is the total size, right and the same is going to hold for all the consecutive threads, for all of them their accessing 2 operands were sitting at a large offset here rather than accessing consecutive operands. So, what is the good thing? Now, see that we are trying to do a load followed by add operation.

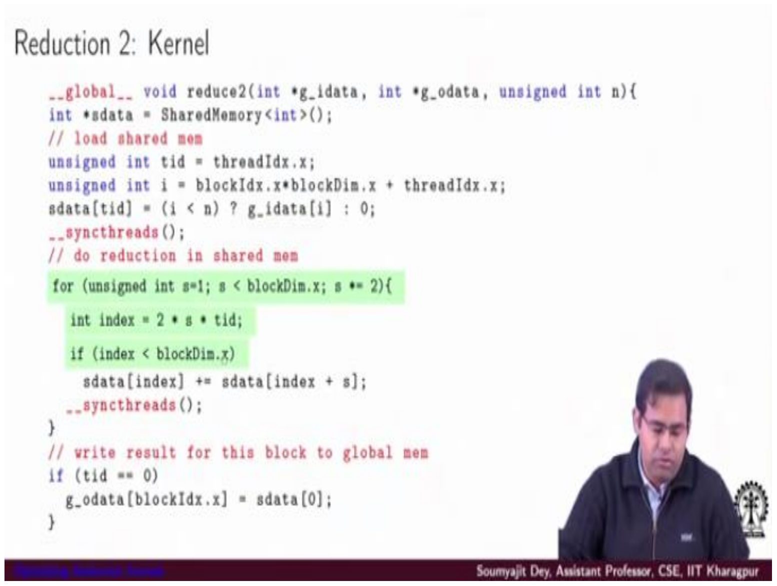
So, again we will just repeat, so if I am looking at the SIMD instruction that will fire; so, all the threads will first load operand1, the thread will load operand 2, the threads will add. So, now if I look at the way the threads are going to load operand1, I have that all the threads are accessing consecutive shared memory banks, all the threads in a warp or accessing consecutive shared memory banks constitutional and that resolves the bank conflict issue which was earlier.

So, just for clarity, let me just repeat again, what happened here in the earlier scenario; so here when all the threads were consecutive ID from the same warp, no divergence but they were loading operand1, they were accessing locations with 1 hop and that would effectively lead to the 2 way bank conflict problem which would occur for both of the loads individually, right but now, when I write a code, which is going to with me this kind of an access pattern.

So, I have consecutive thread IDs inside the warp and they are all loading the operand1 from the shared memory without any bank conflict because all the threads are accessing all the banks in parallel, right so, that is the good thing. So, this load will have no bank conflict, the second load where it is going operand 2, that also is not going to have any bank conflict because again, they are all accessing consecutive banks.

So, I have removed the bank conflicts of both the loads and then I have the unusual addition, so but now the question is; how do I write the code, so that I can make it behave like this?

(Refer Slide Time: 08:20)



Reduction 2: Kernel

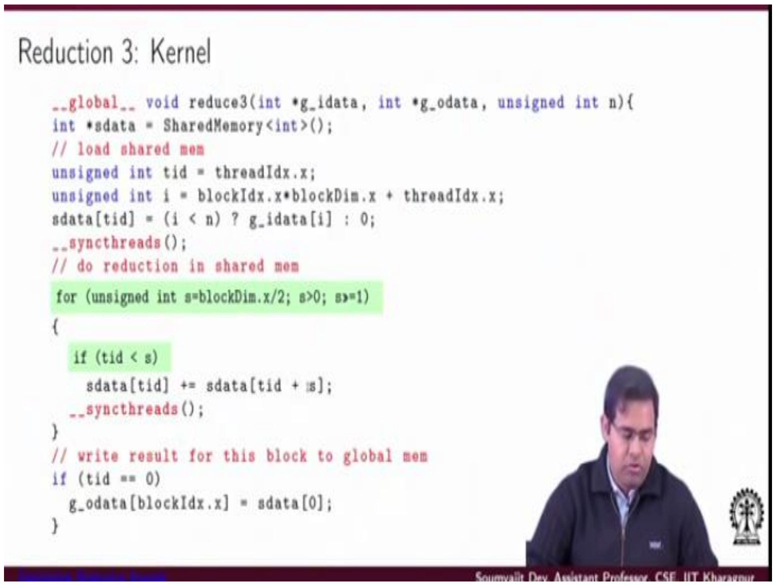
```
__global__ void reduce2(int *g_idata, int *g_odata, unsigned int n){
int *sdata = SharedMemory<int>();
// load shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = (i < n) ? g_idata[i] : 0;
__syncthreads();
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2){
  int index = 2 * s * tid;
  if (index < blockDim.x)
    sdata[index] += sdata[index + s];
  __syncthreads();
}
// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, observer what was the earlier code; so the earlier code started with a stride of 1 that is why the threads were loading consecutive IDs in the first iteration, then it was loading with a stride of 2, in the second iteration then stride of 4, in the third iteration so on and so forth, right. So, this was the kind of code that we are executing but now, if I want this kind of an access pattern that means, I want all the operand ones of consecutive threads to be also consecutive or operand 2 are consecutive thread to be also consecutive.

Then, I need each of the threads to start with the highest stride and then in the next iteration, the stride should reduce, right. So, here this is like; the stride is half of the total array size and then it should reduce by half and then it should further reduce by half like that which would meant I do not count from the lower value of stride to the higher value in multiples of 2 rather I start the loop with a highest value of the stride and get down, each time dividing the strides factor by 2.

(Refer Slide Time: 09:45)



Reduction 3: Kernel

```
__global__ void reduce3(int *g_idata, int *g_odata, unsigned int n){
int *sdata = SharedMemory<int>();
// load shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = (i < n) ? g_idata[i] : 0;
__syncthreads();
// do reduction in shared mem
for (unsigned int s=blockDim.x/2; s>0; s>>=1)
{
  if (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

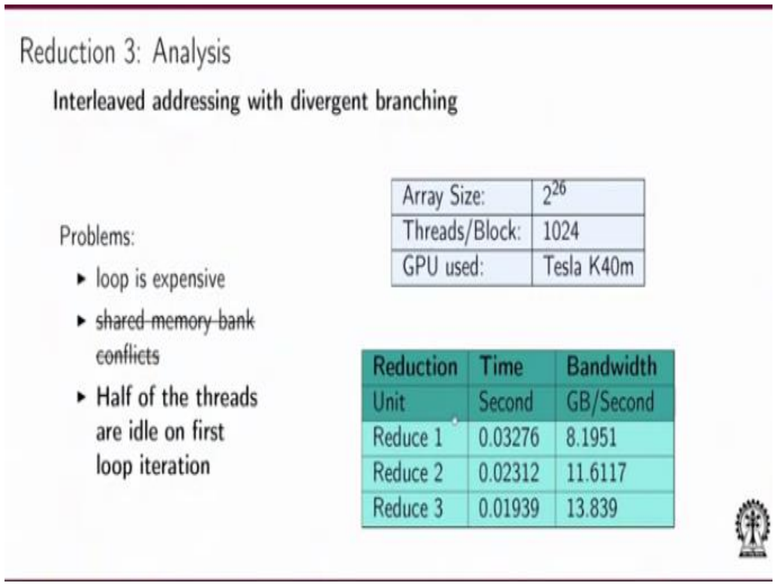
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, if we continue like that, then this is what we get right, which was the original reduction code, now I start with the highest stride which is half of the block dimension and in each iteration, I reduced it by 2 right, inside all we are doing is we are checking whether that the ID is less than the stride because of course, I have started with the highest stride value, so every tid that is going to really get into the loop will be less than the stride value as you can see that here the stride value is this much.

So, the tid's that are going to get into the loop should be up to this, in the next iteration, the stride value is this, the tid is that should be active should be 1 less than this, so on so forth, right. So, in this way we keep on continuing and the loop will keep on executing, decreasing the stride value and inside each iteration, I have got; I mean, all the threads, getting half in number but I also have in each iteration, the threads who are consecutively idle in a warp, they do not diverge.

And they also do not get into any shared memory bank conflict, I hope this is clear without example, yeah, sorry this is, was the; earlier was reduction 2, this is reduction 3. So, I have the; yes, that is the stride value starting from the highest stride keeps on decreasing right, keeps on decreasing and inside the loop, every thread is doing the usual sum and then again in the loop, you decrease the stride and again, getting the loop, you count the sum with the stride decreased and you keep on doing this.

(Refer Slide Time: 11:25)



Reduction 3: Analysis

Interleaved addressing with divergent branching

Problems:

- loop is expensive
- ~~shared memory bank conflicts~~
- Half of the threads are idle on first loop iteration

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839

Now, let us look an analysis of this code, so as you can see that there is gradual reduction in execution time, so here we have given some statistics considering an error size of 2 to the power 26, thread blocks of size 1024 which means, I am calling the kernel multiple times and doing the reduction, using my reduction kernel, 1, 2 or 3 whichever in each iteration, it is reducing the set of blocks and with requisite call of the kernel, finally I get to 1 value, right.

So, the reduction code when it reduces this entire array for tesla K40 GPU as you can see these are the timings and these are the bandwidths so, the timings tell me the total execution time considering these initial array size, so of course you can understand the timings with respect to multiple kernel invocations, the number of kernel invocations are required for reducing the entire array and this bandwidth gives me the total number of loads and stores that have happened divided by the amount of time spent.

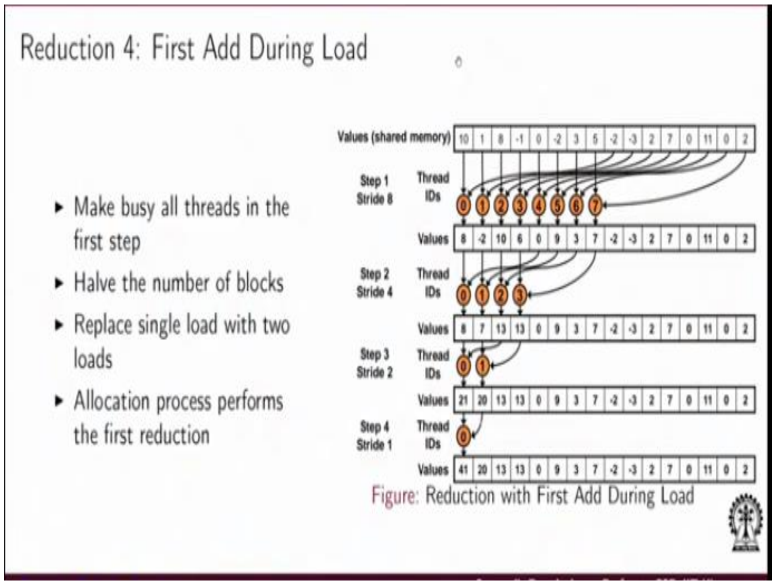
So, what is the CB per second bandwidth consumed from the memory right, so as we can see with reduction of conflicts for the shared memory, I have higher bandwidth and also I have a reduced execution time as an effect of that and primarily, all the acceleration that is coming in is due to the increased memory band width exploited by the code and that actually is effecting the timing and reducing the timing here, right.

I hope this is clear, just to repeat again, the reduced timing that I am getting is basically due to the increase in memory band width that the program is able to exploit, right. Now, still I would say that we have certain issues, what is that? First of all, this loop is still very expensive again, why do we call a loop as expensive; just to we clear from our earlier discussion on basic architecture, every time your threads get into the loop, you have to check for the loop conditions, right.

So, that is one branching check that you have to do before the threads have getting inside the loop, the good thing here is we have removed the module operation and also the most important thing is half of the threads are idle on loop; on first loop iteration that means initially, after loading all the data from the global memory to the shared memory in every launch of the kernel, only half of the threads get inside the actual loop, right.

So, this is the code that every loop is executing, what is essentially doing is essentially; this is a line of code that every loop executes, apart from just computing the global ID using the block dimension, block ID and thread IDs and here as you can see that we are just doing the load from the global memory to the shared memory, after that immediately, I am using, I am starting to use half of the active thread that I have inside the loop, right.

(Refer Slide Time: 14:34)


Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1 Stride 8
Thread IDs 0 1 2 3 4 5 6 7
Values 8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 2 Stride 4
Thread IDs 0 1 2 3
Values 8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 3 Stride 2
Thread IDs 0 1
Values 21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 4 Stride 1
Thread IDs 0
Values 41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Reduction 4: First Add During Load

- Make busy all threads in the first step
- Halve the number of blocks
- Replace single load with two loads
- Allocation process performs the first reduction

Figure: Reduction with First Add During Load

As an alternative, just to increase the part thread activity, what I can do is; I can increase the amount of computation that every thread has to do in the first iteration, as you can see in the first iteration up to reduction 3 for method; in all the methods; reduction 1, reduction 2, reduction 3, there is not much activity to be done by each of the threads apart from the loads, right after for the first; for one half of the threads.

All they are doing is just the loading the shared memory and after that half of the threads do not have any activity, instead of that what you suggest is; we give more warp for each of the threads by making them do some more additions, right. So, make busy all threads in the first half and half the number of blocks. What do we mean is; let all threads not only load data from global memory to the shared memory.

But all threads also do some reduction by themselves for example, okay maybe I make each thread load 2 data from the global memory and then add them and then stored into the shared memory that is increasing the part stride activity, as a side effect that is going to half the number of blocks also, right. So, I increase the part thread activity, I am as the side effect, decreasing the number of blocks.

So, essentially I am replacing one single load with 2 loads and in the addition for all the threads right, so this is the first step of the reduction which we ensure that it is going to be done by all the threads together. So, I mean from mathematical point of view, what is happening is you have too many threads that you have launched, you have launched too many threads here and then you are certainly increasing, I mean you are using only half of them.

So, if I am trying to draw a picture representation here, so let us say launching these many threads, all of them just do the global load and then half of them keep on working, right. So, if I plot part thread activity, this is how things are really happening right, now with this step what are we essentially doing? So, then, okay let me keep the original picture here, so with this step, we are making it like this, right because I have increased the part thread activity here.

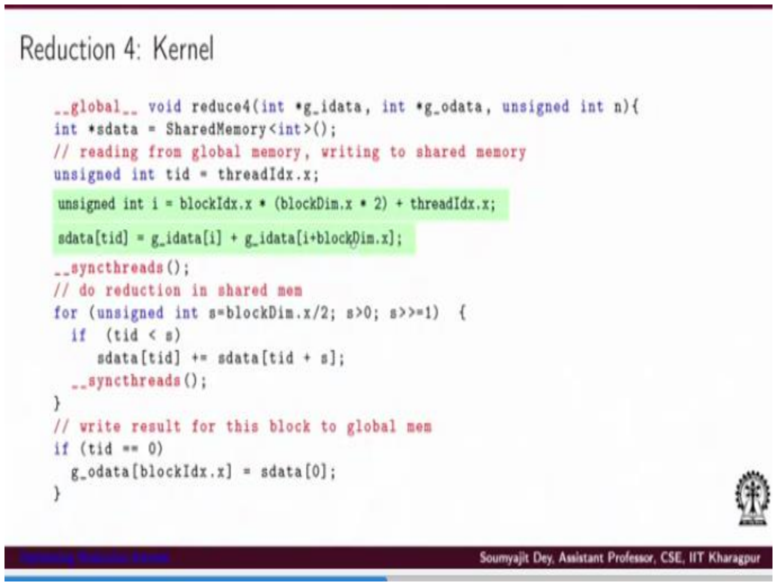
And also sorry, this should be a bit yeah; so what I am now doing is okay, I am launching less number of threads and this threads are only taking care of this part of the activity as well as the other activities right, so I am launching this many number of threads and they are taking care of the activities which earlier this half of the threads were doing, now I do not launch them, an increase the overall part thread activity by launching half number of threads and make them do more of initial work here.

So, that is the important point we are trying to make that if we can find a nice balance between part thread activity and the number of threads that is a good way to look into a issue of parallelism. So, just to explain further, I have got some activity to be executed and I have launched too many threads and I am not executing all the threads throughout the lifetime of the activity, so that does not make sense.

Rather than that I should try to do a load balancing here, I increase the amount of work to be done per thread and decrease the number of thread, then since and I am using less number of threads and utilising them more, I will have less execution time because I also put less overhead on the GPU schedule and other resources in terms of context saving and doing the scheduling job of a higher number of threads, right.

So, the good thing is I do not use too many threads which are idle, for most of the time rather and in that way I save time in saving context and scheduling.

(Refer Slide Time: 19:41)



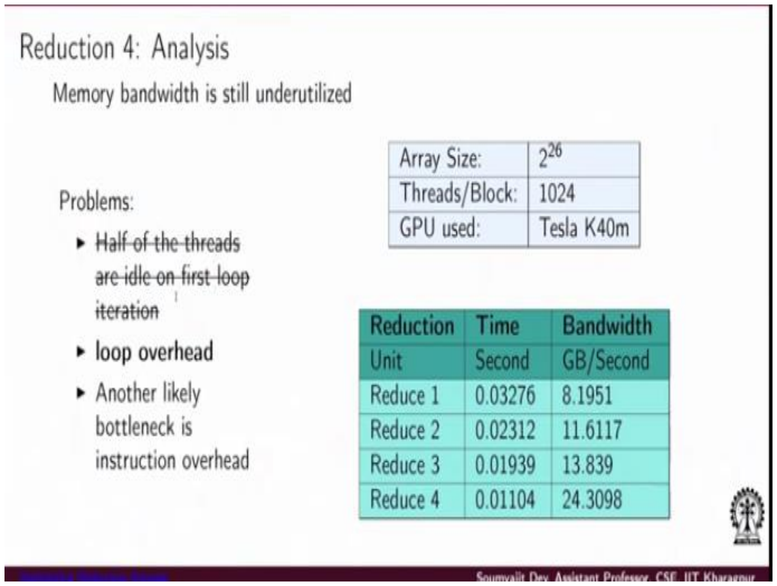
Reduction 4: Kernel

```
__global__ void reduce4(int *g_idata, int *g_odata, unsigned int n){
int *sdata = SharedMemory<int>();
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
// do reduction in shared mem
for (unsigned int s=blockDim.x/2; s>0; s>>=1)  {
  if  (tid < s)
     sdata[tid] += sdata[tid + s];
  __syncthreads();
}
// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, then if this was my original reduction 3 kernel, just have a look, this is the part of code, which is now going to change and it instead will become just like an earlier, it was just, you load 1 global data point and store into 1 shared data point. Now, each thread is loading 2 global data points and they are storing into the shared memory, right that is all that is happening.

(Refer Slide Time: 20:07)



Reduction 4: Analysis

Memory bandwidth is still underutilized

Problems:

- ~~Half of the threads are idle on first loop iteration~~
- loop overhead
- Another likely bottleneck is instruction overhead

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

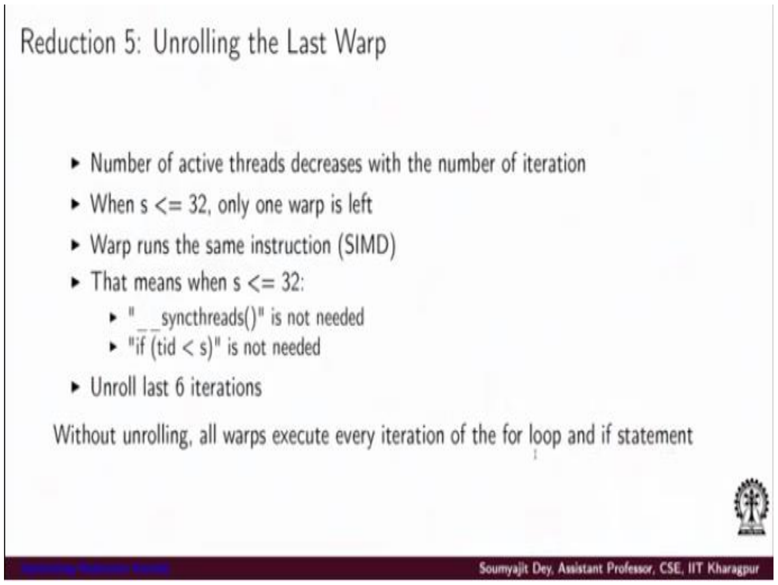
Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.3098

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, the analysis would be like this that I have just increased the part thread activity due to the; I have a tremendous increase in effective bandwidth usage and other result, further reduction in the execution time. So, I do not have this problem of half of the threads going idle on the first loop iteration now, but I still have this issue of the loop over it because every time I launched threads eventually, getting to the loop and they have to in every iteration of the loop, they have to check the loop conditions and all that.

And now, the another likely bottle neck can be the instruction overhead because now, I am executing more instructions per thread but I can see that that is going to be a nice balance here.

(Refer Slide Time: 20:50)



Reduction 5: Unrolling the Last Warp

- Number of active threads decreases with the number of iteration
- When s <= 32, only one warp is left
- Warp runs the same instruction (SIMD)
- That means when s <= 32:
  - "__syncthreads()" is not needed
  - "if (tid < s)" is not needed
- Unroll last 6 iterations

Without unrolling, all warps execute every iteration of the for loop and if statement

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, the 5th reduction version is about the observation that I do not need this kind of synchronisation barrier to be put in, when I am executing the last warp that means, when the number of active threads reduce to 32 then, what happens; then as we can seen that we already removed the divergence from the reduction 1 that means, the threads that are active they are now with consecutive IDs.

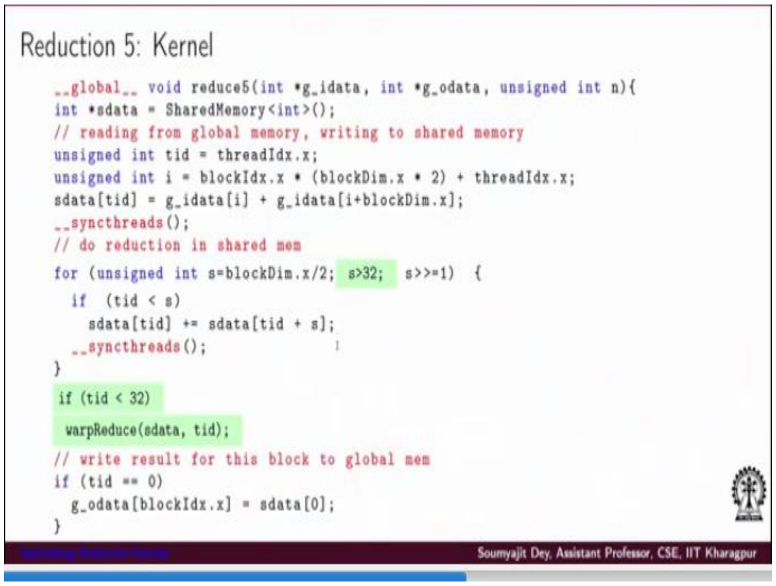
So, they actually form a warp, right so, since this threads are themselves forming a warp and we know that warps execute instructions in lockstep from the programming point of view and that would actually make these sync thread operations redundant here, so this is not needed and moreover, this check will also not be needed that whether the tid is less than s is also not needed.

Because whatever the threads are there, I mean which are going to warp, I mean the other threads are really not going to do any effective warp, right. So, because I am final interested with only the thread ID 0, which will give me the sum, so at this point, I do not need any of them. So, the good thing I can do is; I can remove the same thread and I can unroll the last 6 iterations, why last 6 iterations?

Because from 32, then 16, then 8, then 4 and like that finally, with 1 so, this last 6 iterations, I can also save into the loop condition checking right, so the good thing would be that I simply remove the loop condition and all if conditions, when the number of threads on which I required to focus changes to 32 and they all come inside a warp, right. So, then I will simply unroll this loop and write similar statements consecutively.

And then I do not need this loop at all for those last 6 iterations, I do not need to check this loop condition, I do not need to do this if else check, so without the unrolling, all warps execute every iteration of the for loop and the if the statement and this is what we can just remove by doing this unrolling operation. So, just recalling what we had in loop 4; this reduction 4 was that we were every time in the loop iteration were checking this condition.

(Refer Slide Time: 23:38)



Reduction 5: Kernel

```
__global__ void reduce5(int *g_idata, int *g_odata, unsigned int n){
int *sdata = SharedMemory<int>();
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
// do reduction in shared mem
for (unsigned int s=blockDim.x/2; s>32;  s>>=1)  {
  if  (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
if (tid < 32)
 warpReduce(sdata, tid);
// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

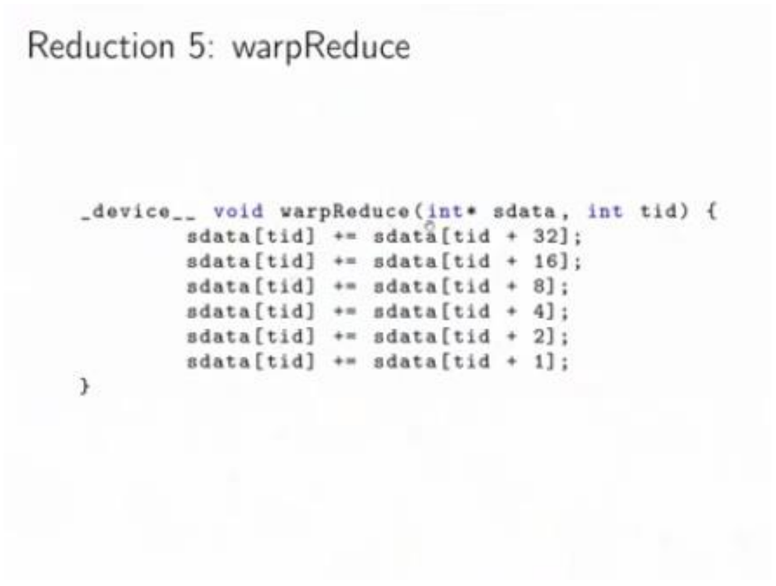
And now, we want to remove it, so the alternate thing that we will do is; earlier this loop was executing up to stride greater than 0, now I make every warp go through this program this for loop, as long as the stride is greater than 32 but once the stride variable decreases to less than 32 okay, once this stride variable decreases to less than 32 for the threads, then I do not let them get into the loop at all and then and I just make a call to this warp reduce function for all threads with the tid less than 32 and just forget the other threads here.

So, as you can see this code will be executed as long as s is greater than equal to 32, once s is less than 32, I completely skip this loop, right. So, now the s is less than 32 and at that point, I just check for the last warp because that is what I'm interested in so, for the last warp, I make a call here to this warp reduce function, fine. I hope this is clear so, earlier what was happening is all threads were executing this loop, right.

We have identified that once I am only at the last few stages of the reduction where I have only 32 consecutive values to warp with and that would mean 32 consecutive threads to warp with sorry, 64 consecutive values and 32 consecutive threads to warp with, I do not need to getting to the loop anymore because I will just write those statements in line one after another avert the loop condition and more importantly, I can avert the sync thread here.

So, I execute this loop only up to s greater than 32 after that I am practically, only interested in the tid's which are less than 32, right because with s equal to 32, what will be interesting is; the first 32 threads working on the 64 consecutive data points which will be handled by this warp reduce function.

(Refer Slide Time: 25:48)



Reduction 5: warpReduce

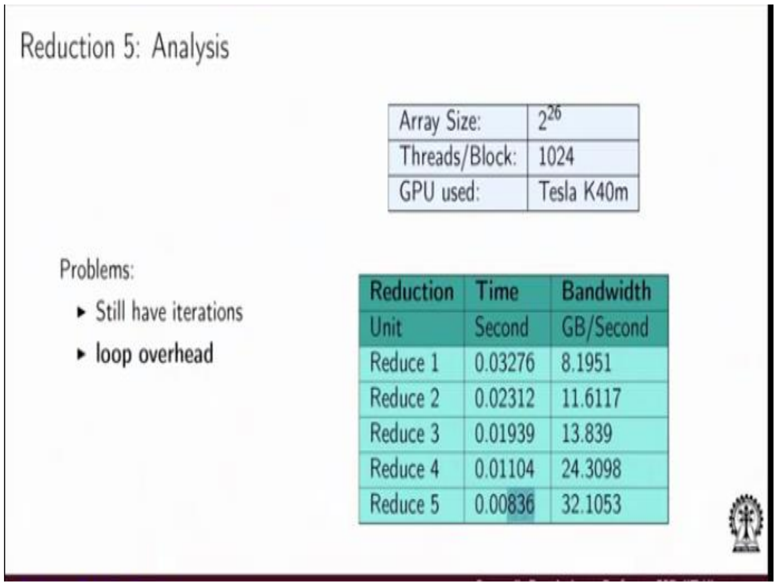
```
_device__ void warpReduce(int* sdata, int tid) {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid + 8];
        sdata[tid] += sdata[tid + 4];
        sdata[tid] += sdata[tid + 2];
        sdata[tid] += sdata[tid + 1];
}
```

And what it will do is; it will just as you can see, I have just copied this body of loop in line one after another with decreasing value of s, so manually I am doing this without executing the loop here, I have completely hard coded 32, 16, 8, 4, 2 and 1, so here after this point, I do not need, I am just repeating this part again, after this point when s is; I have executed the loop with s equal to 64.

After that point, I am only interested in the first warp, I leave all the other warps, I allocate the first warp to execute this sequence of statements, right and as you can understand clearly now that this will be the thing that I am only interested in, I do not care about the other warps now and so, when I execute this sequence of statements, the advantage I gain is; I do not have to execute this sync thread, I do not have to execute this for loops condition check and also the if condition check, right.

So, that would speed up the execution here and then and I just use the final thread to do the final sum, I mean to report the final sum from this data 0 to g data, the output data point in the global memory, right.

(Refer Slide Time: 27:09)



Reduction 5: Analysis

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Problems:

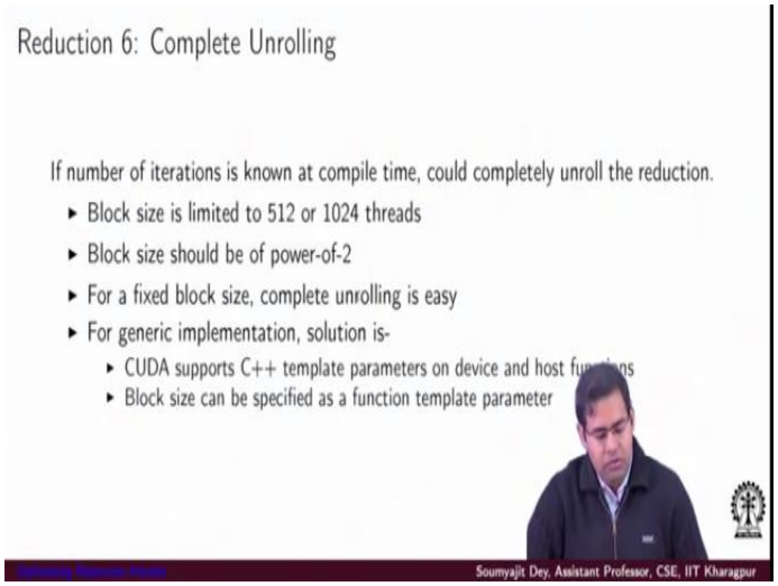
- Still have iterations
- loop overhead

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.3098
Reduce 5	0.00836	32.1053

So, if I do an analysis here, then as you can see that this is further doing an order of reduction in the timing by unrolling the last warp, so this has been a quite a successful optimisation again and this was also the result of increasing the band width further, I mean effective increase in the band width that we are getting for the same code, same array and same thread block choice but still I have iterations and loop over it.

Because although, this has made the last warp go find but still for the previous warps, I have this issue of loop overhead and also I have this multiple iterations to execute right.

(Refer Slide Time: 27:52)



Reduction 6: Complete Unrolling

If number of iterations is known at compile time, could completely unroll the reduction.

- Block size is limited to 512 or 1024 threads
- Block size should be of power-of-2
- For a fixed block size, complete unrolling is easy
- For generic implementation, solution is-
  - CUDA supports C++ template parameters on device and host functions
  - Block size can be specified as a function template parameter

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

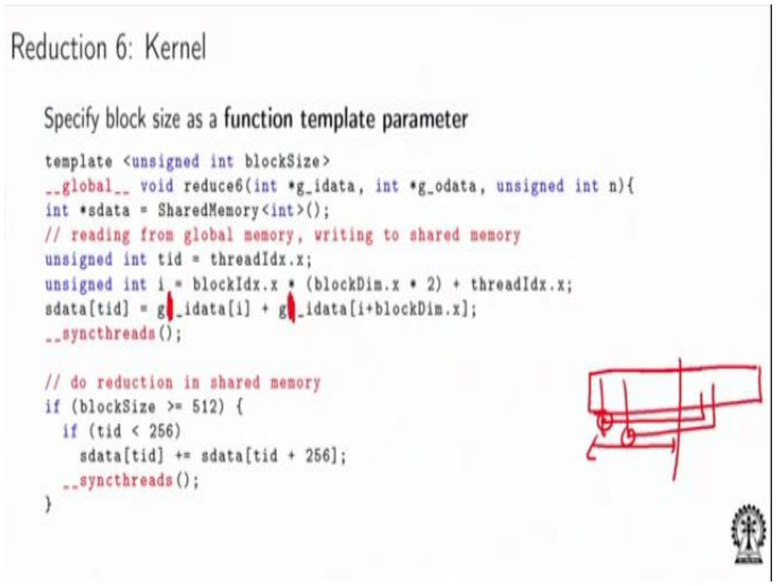
So, what can be an alternative here; the alternative here is that if I can completely unroll the execution, I can remove the loop completely but the issue is why do I really need the loop; I need the loop that because at the compile time, the number of iterations required is unknown, right because I do not know the block size, the number of threads on which reduction kernel is going to be applied, right.

Now, if I take some static decisions that let us say, I limit the block size to 512 or 1024 threads, right because anyway, it is going to be power of 2. So, if I know the block size, then doing a complete unrolling of the code is going to be easy, right so, in this case what we can do is; we can take an advantage of C++ templates and that is something that CUDA is going to support. So, without getting into too much detail of what is a template and all that, we will just go through the example.

Of course, this is something advance, so we will just give you an example here, so I mean you can consider this addendum to our original syllabus material, I mean this is just for the people who are more interested into the fundamentals of I mean, programming and further optimisations here, so instead of passing the block size as a I mean, making an implementation inside which I am handling a generic block size.

Let us assume that the block size can be specified as the function parameter, right and or more I would say, is like the template parameter.

(Refer Slide Time: 29:29)



Reduction 6: Kernel

Specify block size as a **function template parameter**

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n){
int *sdata = SharedMemory<int>();
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();

// do reduction in shared memory
if (blockSize >= 512) {
  if (tid < 256)
    sdata[tid] += sdata[tid + 256];
  __syncthreads();
}
```

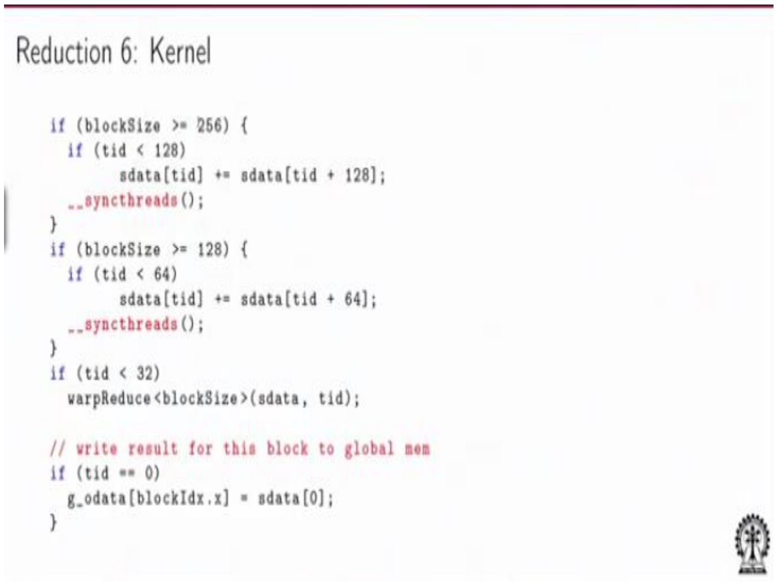
What is the good thing then? So, instead of having just the reduction kernel, I make a template of it with the parameterised block size. So, what is the advantage? You just tell what is the advantage of function template here, so if I have a template declaration here with parameterised block size followed by the usual reduce kernel, then the system will make multiple implementations of this function but different possible block sizes here.

Actually, I mean, I am adding more dynamism here right, so essentially as you can see that if I decide that okay, the block size is limited by the size 512 or 1024 threads, then here, since I am creating a function template here, so what I will do is; I will now to the reduction entirely through sequence of because if else blocks, completely remove the loop, so this part is usual, you can just consider this as an computation of the global ID.

And then, okay I think we have some, we can remove this, yeah, so you are computing these here the s data thread ID locations and then you do the reduction in the shared memory using a sequence of if else blocks, so you check what is the block size and based on that you get into the corresponding part, where you are starting to do the reduction. So, if the block size is greater than 512 billion, then you will execute this block, width threads that are of ID up to 256, I mean less than 256, right.

Because then you have a block of size 512 and you are going to use half of the threads to do the iteration operations right, like that so on so forth.

(Refer Slide Time: 31:55)



Reduction 6: Kernel

```
if (blockSize >= 256) {
  if (tid < 128)
        sdata[tid] += sdata[tid + 128];
  __syncthreads();
}
if (blockSize >= 128) {
  if (tid < 64)
        sdata[tid] += sdata[tid + 64];
  __syncthreads();
}
if (tid < 32)
  warpReduce<blockSize>(sdata, tid);

// write result for this block to global mem
if (tid == 0)
  g_odata[blockIdx.x] = sdata[0];
}
```

So, that I show it is going to continue, so essentially we are just unroll the loop and we have just written a sequence of if else statements, where the statement block will get activated depending on the block size so essentially, the block size is being provided as a template parameter here, so that you have different versions of the code and you have based on the block size, the code will actually execute whichever is applicable, right.

So, suppose your block size is 512 that would mean, your code will start executing and will execute this entire sequence of if blocks, right because that is what the loop will do; the equivalent loop will do, right. So, essentially with 512, you have how many in each iteration?

You have got the; so essentially, when to compute with strides which is starting with 256 because block size 512, stride would be 256 and with that we start the reduction.

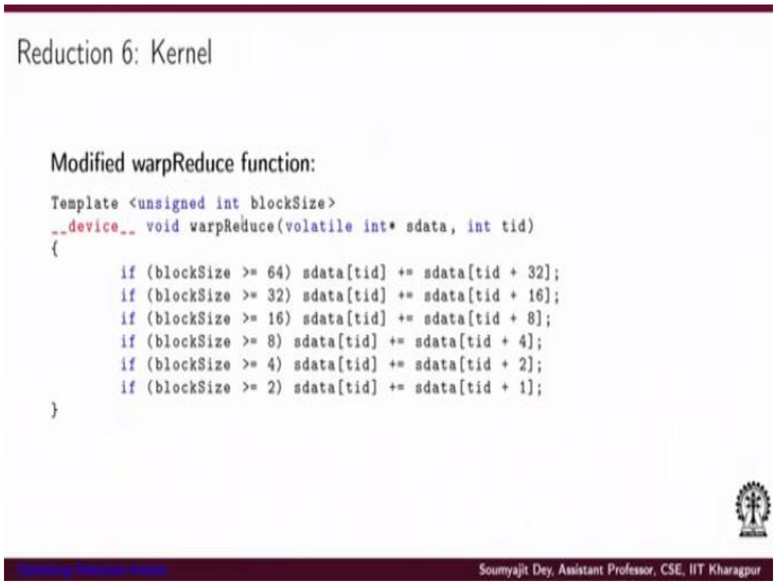
Just that code as you can see in every step, the reduction; reduction stride size will decrease, all we have done here is we have simply replicated their behaviour without the loop, right, so we just put in the if else blocks, so each block size is greater than 512, I start executing from this block. If the block size is not greater than 512 but greater than 256, greater than or equal to, then I just start writing it from this block, right.

And why do I have this reduction by half because of the nature of the code, we know that the block size will be multiples of 2 and in every iteration, it will be parts of 2 and in every iteration, you are going to divide the stride by half, so just have a closed loop into our earlier reduction kernels, look at the for loops and mentally, unroll the for loop with the reducing value of strides and starting from the 5th block size and this is the execution sequence which was suppose to get.

All we are doing is; we are removing the loop, so that you are removing the loop chip and you are just executing the internal behaviour of the loop here, right because as you can see everything else is same, at the end you again reduce the last warp but again you are using function template parameter here.

So, as I am repeating here for our course purpose, we do not assume that you will have knowledge of C++ or function templates, considered this as just some extra interesting piece of code, where we show that how using this function templates, you can make good use of; I mean you can actually make good use of this kind of templates to do further optimisation with respect to reductions.

(Refer Slide Time: 34:52)



Reduction 6: Kernel

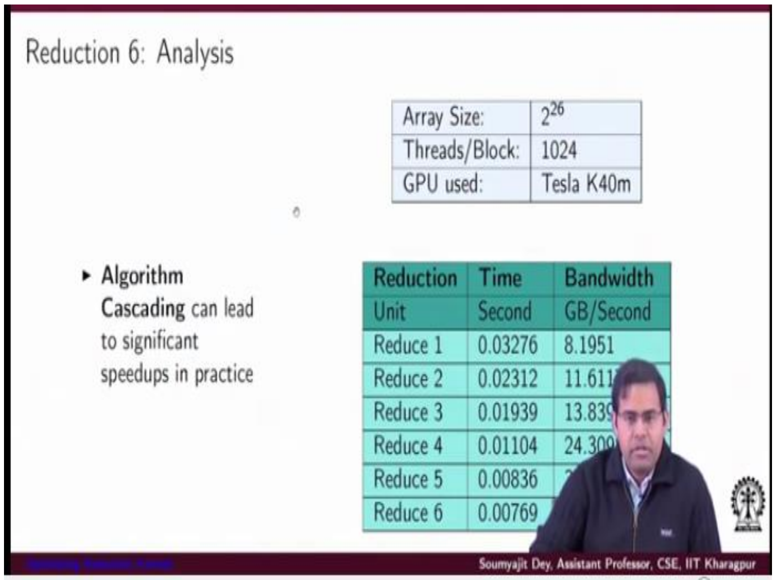
Modified warpReduce function:

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid)
{
        if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
        if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
        if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
        if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
        if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
        if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, this is the corresponding way when you are going to write a warp reduce function as in a template from, right. It is just the same but all we are doing is you are essentially creating different versions of the function for different possible block sizes, right.

(Refer Slide Time: 35:13)



Reduction 6: Analysis

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

- Algorithm Cascading can lead to significant speedups in practice

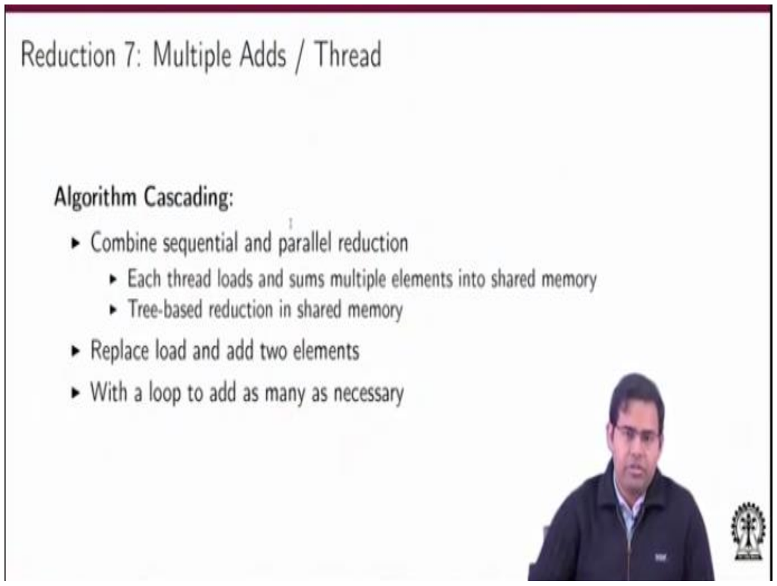
Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.611
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.309
Reduce 5	0.00836	[illegible]
Reduce 6	0.00769	[illegible]

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, if you do analysis here as you can see that since this approach is able to remove the loop and just unroll the execution of the loop as the sequence of if else blocks, the sequence of if blocks, it leads to further reduction in the execution over it, right. So, I hope this is a bit clear to you, so just remember the point to be taken in this case here is the issue with unrolling is; I do not know what is my start size is.

So, instead of writing a function, I write a function template where I give the start size is the parameter right, so this can father we optimised with significant speedups in practice.

(Refer Slide Time: 36:04)



Reduction 7: Multiple Adds / Thread

Algorithm Cascading:

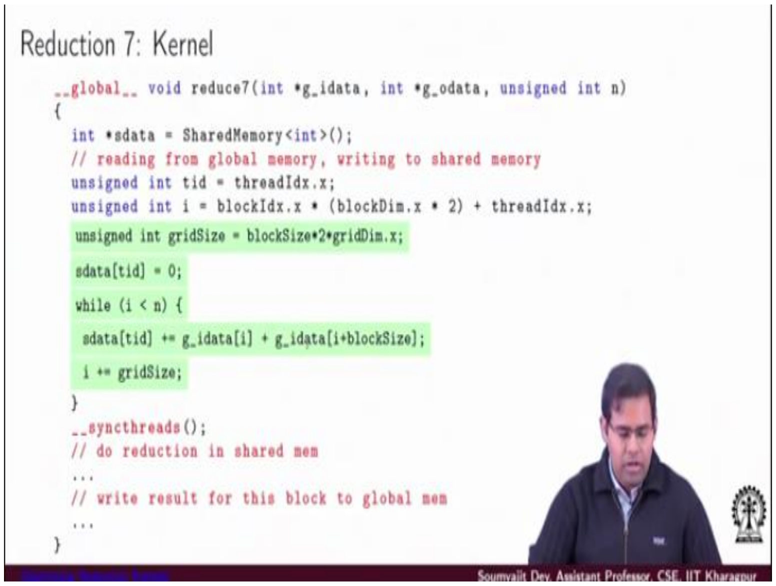
- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory
- Replace load and add two elements
- With a loop to add as many as necessary

And the approach would be known as algorithmic cascading, so let us try and understand what is this issue of algorithmic cascading, now earlier what we saw was that to increase the part thread activity, we just included some activity for all the threats initially, so we did some addition on the global memory operands before doing; before storing them in the shared memory and then proceeding with half of the threads but that part is also configurable, right.

So, I can actually do a combination of sequential and parallel reduction, so each thread instead of loading and summing just 2 elements, it can do more activity, it can load and sum a sequence of elements and then it can follow the idea of tree based reduction that we showed in shared memory. So, all we are trying to say here is instead of just doing 1 add and then storing the data in the shared memory, so this as you can see that this as a part thread activity.

If we do not do this, then what do we really have; we have half of the threads working with the addition and only initially, all the threads were active in doing the load, with these I increase part thread activity by doing one at globally on the data right, so this is the sequential addition performed by each of the threads on the data global; on the global data before storing it into the shared memory.

(Refer Slide Time: 37:34)



Reduction 7: Kernel

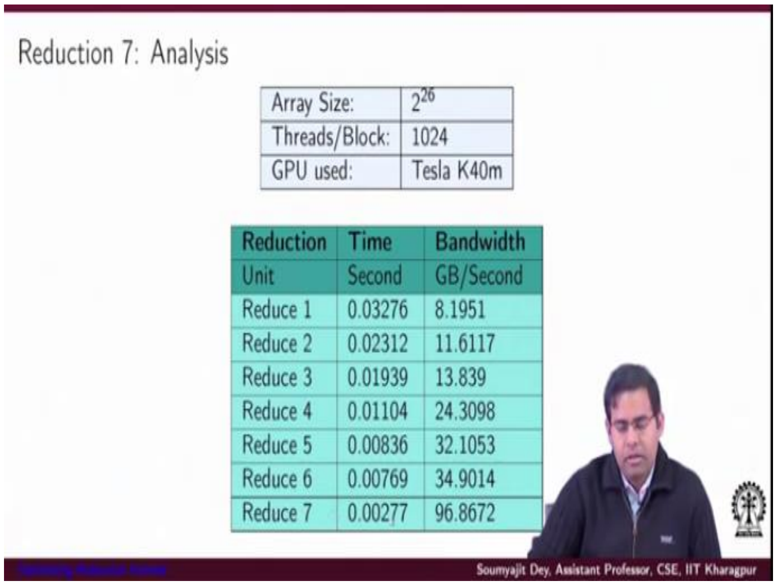
```
__global__ void reduce7(int *g_idata, int *g_odata, unsigned int n)
{
  int *sdata = SharedMemory<int>();
  // reading from global memory, writing to shared memory
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
  unsigned int gridSize = blockSize*2*gridDim.x;
  sdata[tid] = 0;
  while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
  }
  __syncthreads();
  // do reduction in shared mem
  ...
  // write result for this block to global mem
  ...
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

I am saying that we can increase the activity here, so that would mean, I can do some reduction on the global memory and then do the rest of the reduction in the shared memory, right. So, here for and of course, since this is a part thread activity, this is going to be sequential, so as you can see there is similar set of code here, this is a; but this is sequential, so each thread is doing computation on the global data that is as working and then it storing here into the shared memory, it is bringing in more data and like that up to the grid size, right.

So, essentially you are going to do; all you are doing is for each thread, you are doing some part of the reduction sequentially in the global memory, so you read form global memory and write to shared memory, earlier we are doing one addition but here instead you are doing a sequence of additions. So, if we start here with the tid, then you compute for this thread, what is the global data is going to bring that data plus its block size and then you make it warp for some iterations with respect to the grid size, right.

(Refer Slide Time: 38:56)



Reduction 7: Analysis

Array Size:	2^{26}
Threads/Block:	1024
GPU used:	Tesla K40m

Reduction	Time	Bandwidth
Unit	Second	GB/Second
Reduce 1	0.03276	8.1951
Reduce 2	0.02312	11.6117
Reduce 3	0.01939	13.839
Reduce 4	0.01104	24.3098
Reduce 5	0.00836	32.1053
Reduce 6	0.00769	34.9014
Reduce 7	0.00277	96.8672

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And you do instead of doing only on 1 addition, you make it do multiple additions here and with that that you increase more amount of part thread activity and that would give you further reduction by this step. So, just if you want to just go through this code here, let us have a look again, so you are using this thread ID from your t IDx right, so then inside your block, what is this tID doing?

So, this thread for its component of the shared memory of course, you have defined the shared memory here, right and then for this thread, for its component of the shared memory is going to do multiple adds here from the global memory as you can see, so you are accessing the ith global location with the offset of a block size, right. So, you are going to add across this off set of a block size.

And then you are going to put it here, right so, just have a look into this, we will come back to this reduction again to get into the intricacy of this reduction for right now, the basic idea will I like to say is that earlier we just outsourced one part thread addition to all the threads for doing a global add, we are just trying to bring in some more sequential activity part thread to doa load balancing here, right.

So, with this we will ending this lecture, thank you for your attention, from the next lecture, we will go into this in a bit more detail and we will do performance analysis of each of the reductions and continue with further topics, thank you.