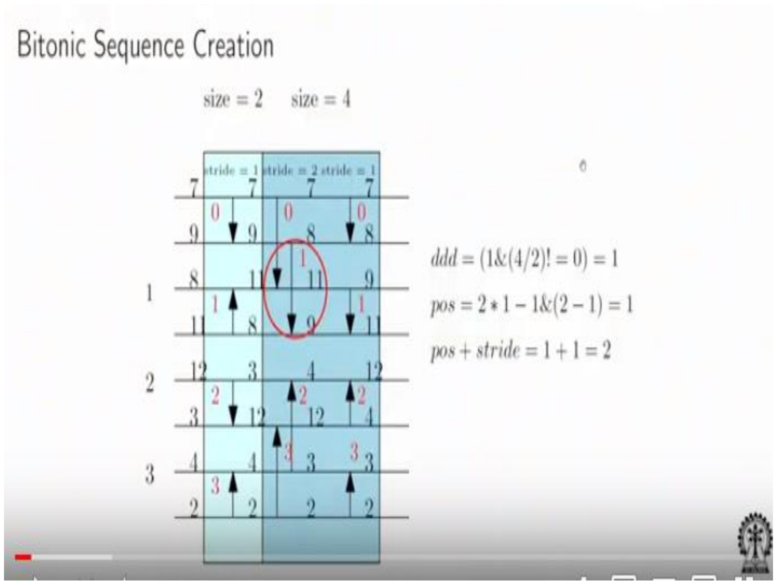
GPU Architectures and Programming

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

Lecture – 33
Optimizing Reduction Kernels(Contd.)

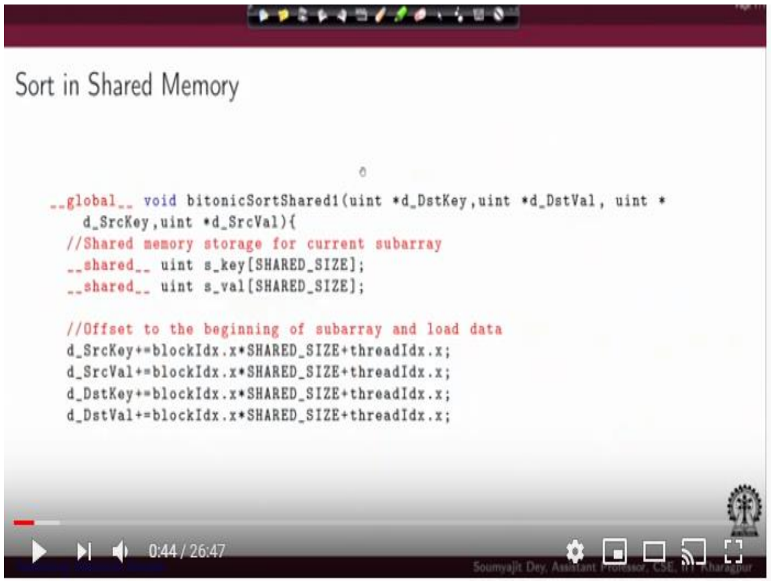
Hi welcome to the lecture series on GPU architectures and programming. So if you remember in the last lecture we have been discussing about Bitonic sort.

(Refer Slide Time: 00:33)


Bitonic Sequence Creation
size = 2
size = 4
stride = 1
stride = 2
stride = 1
$ddd = (1\&(4/2)! = 0) = 1$
$pos = 2*1 - 1\&(2-1) = 1$
$pos + stride = 1 + 1 = 2$

And our discussions progressed up to the part where we are generating the Bitonic sequence.

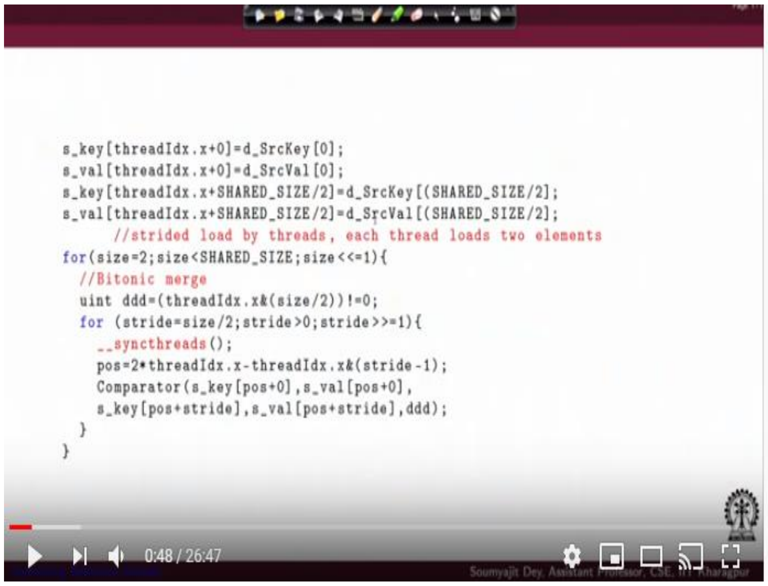
(Refer Slide Time: 00:42)


Sort in Shared Memory

```
__global__ void bitonicSortShared1(uint *d_DstKey,uint *d_DstVal, uint *
    d_SrcKey,uint *d_SrcVal){
  //Shared memory storage for current subarray
  __shared__ uint s_key[SHARED_SIZE];
  __shared__ uint s_val[SHARED_SIZE];

  //Offset to the beginning of subarray and load data
  d_SrcKey+=blockIdx.x*SHARED_SIZE+threadIdx.x;
  d_SrcVal+=blockIdx.x*SHARED_SIZE+threadIdx.x;
  d_DstKey+=blockIdx.x*SHARED_SIZE+threadIdx.x;
  d_DstVal+=blockIdx.x*SHARED_SIZE+threadIdx.x;
```

And a just a small recall. So this was the CUDA code we were showing for the Bitonic sequence generation.

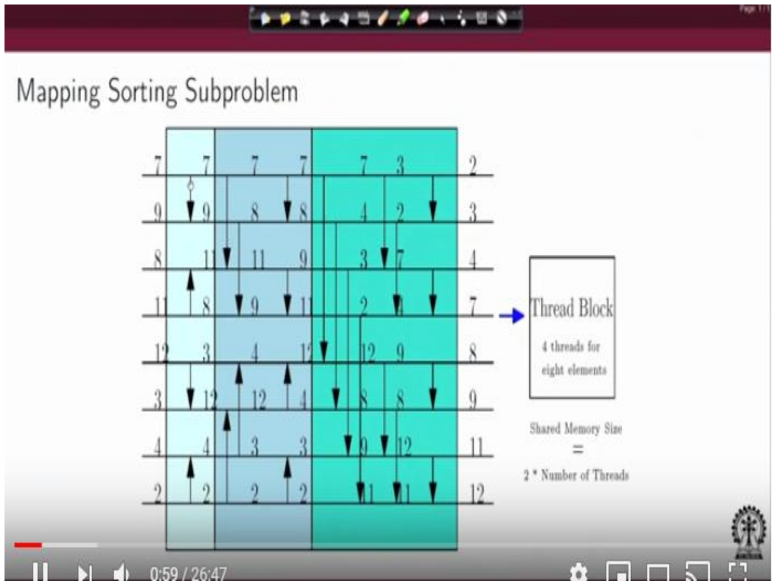
(Refer Slide Time: 00:47)



```
s_key[threadIdx.x+0]=d_SrcKey[0];
s_val[threadIdx.x+0]=d_SrcVal[0];
s_key[threadIdx.x+SHARED_SIZE/2]=d_SrcKey[(SHARED_SIZE/2];
s_val[threadIdx.x+SHARED_SIZE/2]=d_SrcVal[(SHARED_SIZE/2];
      //strided load by threads, each thread loads two elements
for(size=2;size<SHARED_SIZE;size<<=1){
  //Bitonic merge
  uint ddd=(threadIdx.x&(size/2))!=0;
  for (stride=size/2;stride>0;stride>>=1){
    __syncthreads();
    pos=2*threadIdx.x-threadIdx.x&(stride-1);
    Comparator(s_key[pos+0],s_val[pos+0],
    s_key[pos+stride],s_val[pos+stride],ddd);
  }
}
```

So each of the threads were responsible for bringing in 2 values into the shared memory. Then each of the threads.

(Refer Slide Time: 00:57)


Mapping Sorting Subproblem
Thread Block
4 threads for eight elements
Shared Memory Size = 2 * Number of Threads
0:59 / 26:47

As we can see from the previous picture here that each of the threads are tasked with the following operations. Like if I look at the thread id 0 then if I try to figure out that what are the values on which that threads id is working? And so we have a nice example here on the 1st part here that so threads id 0 is bringing 2 values and in the 1st phase of operations thread id 0 is doing a comparison here whereas thread id 1 is doing a comparison here. Thread id 2 is doing the next comparison like that.

All of these are happening in this constant I mean in this with the variables of size =2 and stride =1. As you can see the stride is 1 here for the comparisons and the size =2 denotes that essentially we are doing a bit unique Merge which size 2 here right? I mean just between 2 elements there that because that would be the first phase of the Bitonic sequence generation creation.

In the second phase again we will have size =4 and we will just apply the same logic right? But now we will try to do it with since we are doing it size =4 then well have 2 strides, stride=2 and stride=1 and in this strides threads id 0 would be tasked with doing these 2 comparison these first comparison here with stride=2.

And then when the iteration of the inner loop goes to the next iteration then thread id 0 would again we will be doing another comparison with a reduced stride of 1. Because in the inner loop as we have discussed earlier the stride is decreasing by half right? It is starting from size by 2. So we start from stride 2 and then we moved to stride=1 and similarly when we started with size =2 the stride was already half to 1 and there was no further progress right?

So the outer loop is controlling the size and the inner loop is reducing the stride starting from size by 2. And in that way I have different phases of the Bitonic sequence creation as we have discussed that recursively these are all basically Bitnoic merge operations at different levels right?

So I mean just as an example as we can see that here we have a small Bitonic sequence getting generated 7 it increases to 8 and 11=9 and then again this is being going to let Bitonic merge stage here in the step 2 right? So in that way we have the above code here which is realizing the Bitonic sequence generation part using these 2 cascade of loops right? So overall I mean of course the lecture slides will be with you.

If you take a closer look you will find as we have discussed many times earlier each threads is loading 2 fellows into the shared memory at an offset of share size by 2. Then each thread gets into this loop casket then each thread in each stage of the loops. So the outer loop is controlling the iterations over size values inner is controlling the number of strides for a specific size value and that defines for what is the part threads activity.

So for 0 id threads this is the activity for the threads with id 1 this is the activity for threads. With id 2 we have again marked activities. So the ids are marked here in red. As you can see similarly for thread with id 3 these are the activities and so on full so forth. Now if you look into the code we also had these variables triple d and position which were actually trying to compute the direction of the comparison and the position at which the comparison would happen.

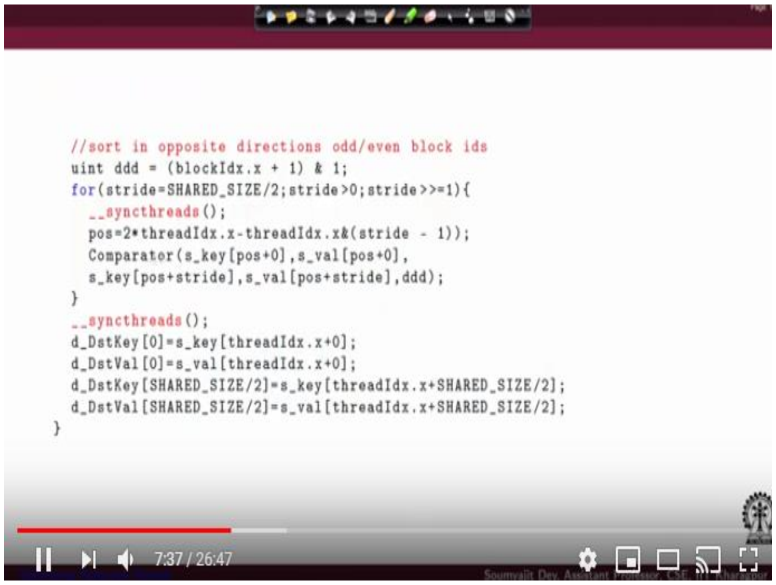
Just to do a check that they are really doing their job. If we take an example here for this specific threads with id 1 in the size = 4 size =2 case that how is this thing really happening? You can see that a this is how the direction is being computed as downwards right? Using that Bitwise and operation. And similarly you can also compute the position and position plus stride locations here.

So just have a look this an example we have tried to mark out for you know coming to the next part of the code assuming that we are okay with this sequence generation idea. Then the other part comes in where we are just trying to do the overall merge right? So the as we can say that the Merge is a more fundamental thing but here what we have to do is we have to do the merge over a large sized Bitonic sequence.

So the first thing is inside the merge phase there will be no change of direction right? Because now my goal is fixed whenever I am applying smaller versions of marge recursively inside the sequence generation phase. Of course there will be a change of direction as it is happening here. So some thread as you can see 0 is always computing.

The comparison side is indication is down for 1 its changes it is like the comparison introduction changes for to the direction. Again changes for 3 there is no change in direction and these are controlled by these variables. But when we come to the overall merge stage it is a global objective right? That you either want to do a completely ascending output or you want to compute a completely descending output right?

(Refer Slide Time: 06:28)



```
  //sort in opposite directions odd/even block ids
  uint ddd = (blockIdx.x + 1) & 1;
  for(stride=SHARED_SIZE/2;stride>0;stride>>=1){
    __syncthreads();
    pos=2*threadIdx.x-threadIdx.x&(stride - 1));
    Comparator(s_key[pos+0],s_val[pos+0],
    s_key[pos+stride],s_val[pos+stride],ddd);
  }
  __syncthreads();
  d_DstKey[0]=s_key[threadIdx.x+0];
  d_DstVal[0]=s_val[threadIdx.x+0];
  d_DstKey[SHARED_SIZE/2]=s_key[threadIdx.x+SHARED_SIZE/2];
  d_DstVal[SHARED_SIZE/2]=s_val[threadIdx.x+SHARED_SIZE/2];
}
```

So the direction is computed once and for all. And then you enter into the loop cascade for doing the different margins for doing the different sequence of comparison operations. Now if we just go back to our original picture here. So this is my merge stage right? So in this merge stage as you can see that we have a validation that directions are all same. We are trying to do an ascending output basically to all the arrows are downward.

We want the Maxwells to all flow down. Now what is happening is we have to identify what is the parts thread activity. So for each threads there will be a downward comparison happening at different stride values and at different data points. So one thing is common the stride value will increase by 2 right? As you can see a stride 4 comparison strike 2 comparisons stride 1 comparison.

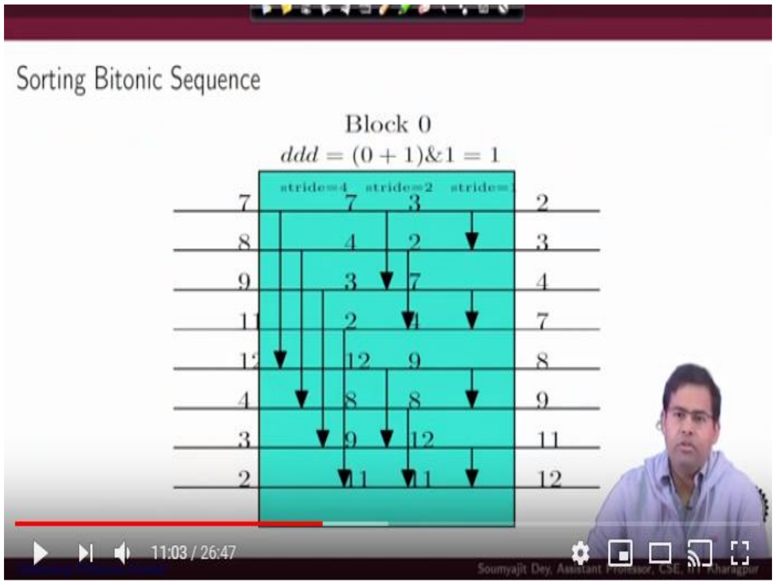
Similarly for each threads right stride 4 stride 2 and strike 1 comparison. The issue is how do we distribute the different comparisons to be done as a part thread activity. So again that is being so this is the loop which is controlling these sequence of stride values for part for each threads right? So the loop iterations for the threads are spread over the X axis in that picture. And here for each threads you are figuring out what is the position. Once you figured that okay I will do a comparison at this stride value for the thread.

The next thing you do is you figure out what is the position of the comparison right? Once you figure out the position of the comparison by this computation. The next thing is you just apply the competitor at that position with that stride and the direction that has been already pre decided right? So with this loop the output will be red in the shared memory. Then what you have to do is since each threads has been taxed with loading 2 values.

Now the threads will also write back to values to the global memory both the keys and the values right? So essentially this the position index computes the location for which the threads will be doing the comparisons like for thread 2 the locations would be here this and this like that. Right in that way for each of the threads the positions would be computed the with different strides the comparisons would happen.

So thread 1 would have these 3 operations. Similarly, 3 comparisons for each of the threads. And finally when the loop iteration ends for all the threads I would have these values in the shared memory which now will be returned back 2 values per thread in the global memory. So we have the example here like so if you look at here this is the Bitonic sequence creation right? So this is a sequence 7,8,9,11 and then 12, 4,3,2 ascending and then descending.

(Refer Slide Time: 09:35)



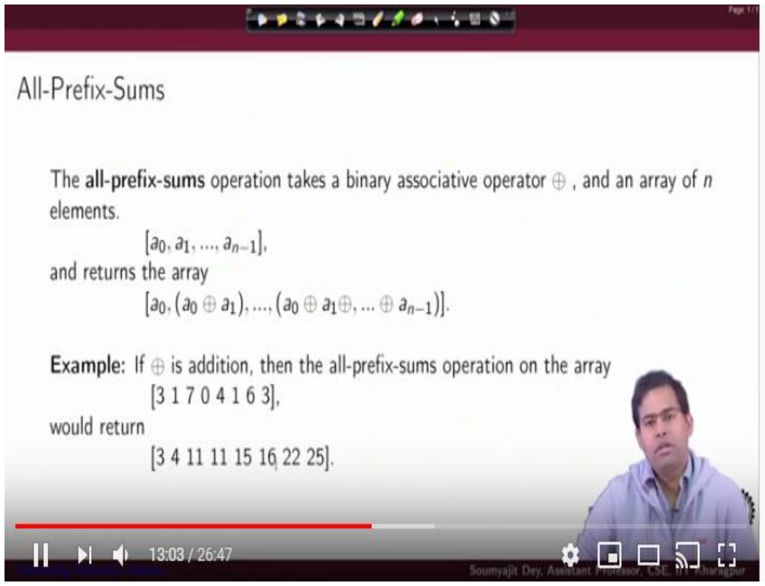
Right now let us look at the merge operations here. So you have these that earlier output as the input and then we are showing that first 4 block 0 that direction has been decided as 1. So everything downwards and then for different stride values. We are just trying to show here as the using the example how the values change. So for thread 0 when these activities is done 7 and 12 nothing will change.

Then it is looking here at 7 and 3 so 7 will come down. And then again its just looking here and it will make 3 come down here and 2 go up in that way with all the other values also being getting swept through the competitors you have the final output at here as output of the sorting network. So that would be our overall discussion on Bitonic chart implemented as a CUDA program.

So this actually gives you a nice overview how to use a sorting network and try to create a parallel implementation of that and that will be useful for you to do some. And you can try with some large sized arrays how to make these implementations and you can study things like how shared memory and other issues can be effecting the computation. I mean there are many interesting issues there.

Like for example how do I really configure the shared memory? Is it really affecting the effecting the computation speed? Should I compare the shared memory as a larger shared memory with a smaller element cache or a smaller shared element cache. What is really going to help me in this case? So these are important questions you may ask yourself. You can try in a CUDA based system for your purpose.

(Refer Slide Time: 11:23)



All-Prefix-Sums

The **all-prefix-sums** operation takes a binary associative operator $\oplus$, and an array of n elements.

$[a_0, a_1, ..., a_{n-1}]$,

and returns the array

$[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus, ... \oplus a_{n-1})]$.

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array

[3 1 7 0 4 1 6 3],

would return

[3 4 11 11 15 16 22 25].

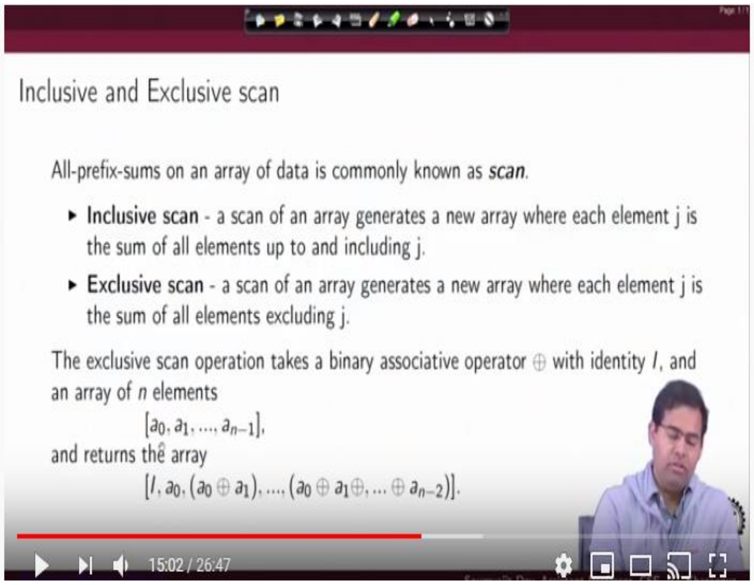
Now we will come to the other example that we have now which is on prefix sum. So to understand what is a prefix sum operation. So essentially is a difference. The difference between reduction I mean parallel reduction that we did earlier and prefix some is that earlier when we were doing parallel reduction you are given the sequence of values and all you are doing is you are only compute only interested in this value right?

So given the operator you are interested as zooming that just does simple addition. It can be any other as associative operator you are interested only in this expression right? But now we are seeing that okay no I also want the other prefixes being operated like this. So essentially I am looking at an output which is a0 followed by a0 then assuming this addition could I am saying it can be any associative operation a0 at 1 and at the last it can be a 0 +a1+a2 like that. And finally the addition of all the numbers right?

Let us assume that this is what I want instead of only computing the total sum right? So in that way if this is the input in a prefix operation should return me this entire sequence right? Earlier when were doing a parallel reduction as human as a simple addition operation I was only interested in this 25 value. But now I am saying that no I want an output at a computer which is containing the sum of all the prefixes.

So in the ith location I have the sum of values from a0 +a1+a2 up to +ai right.

(Refer Slide Time: 13:07)



Inclusive and Exclusive scan

All-prefix-sums on an array of data is commonly known as *scan*.

- **Inclusive scan** - a scan of an array generates a new array where each element j is the sum of all elements up to and including j.
- **Exclusive scan** - a scan of an array generates a new array where each element j is the sum of all elements excluding j.

The exclusive scan operation takes a binary associative operator $\oplus$ with identity I, and an array of n elements

$$[a_0, a_1, \ldots, a_{n-1}],$$

and returns the array

$$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus, \ldots \oplus a_{n-2})].$$

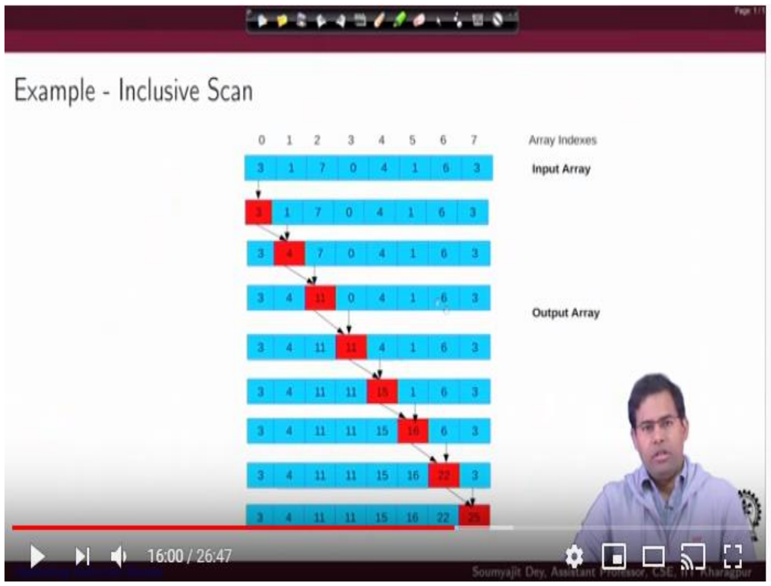
Now the issue is how can these be accelerated? Can the usual technique that we applied for computing the normal parallel reduction is it directly applicable here? What are the waste we accelerate and all that? Now one obvious question is here that we can have 2 possibilities. One is inclusive scan and Exclusive scan. So inclusive scan would mean that that when you are producing the output array then at any point any jth location in the output array you are interested in the sum from a0 up to aj.

Whereas if I say it is an exclusive scan you are saying that okay you are interested in a0+al like that up to aj-1. So essentially we are seeing that okay whether I will include the element in that location in the sum or whether I should exclude it and I will just put there the value which is for the location up to up to the previous one. So if I am doing an exclusive scan then considering this a0 up to an-1 as my input the output that I am looking at is at the initial position I am not having a0.

What I am having is i which is the identity of the operation that is being given here. Considering that the operation is addition the identity would be as 0. So I would write as 0 here in general I am just reading the identity as the operator I the identity of the operator under question. So here considering addition I will have 0 in the next location. There a 1th location I am just having a0 in the second location I am having a0+a1.

So in the second location I am having the some of the first 2 elements in the third location I am going to solve the first 3 elements and so on so forth. Right whereas if I have done an inclusive scan in the second location I have I would have put a0+a1+a2 like that.

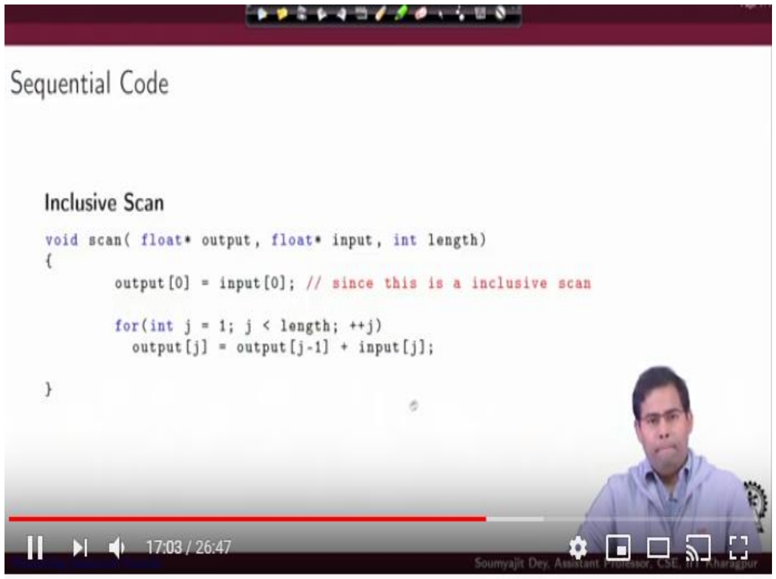
(Refer Slide Time: 15:18)



So this is a example by figure that if Im doing an inclusive scan so this is the situation for an exclusive scan right now what would be the situation for an inclusive scan? By the way I mean you may be wondering why we are suddenly using this word scan here because in general this operation is popularly known as a scan of the data under question right? A scan of an array so when I say it is an inclusive scan the picture like ideally we are looking like this.

Suppose this is your input array and your output array is being computed through the stages like this that 3+1 is 4 and 4+7 is 11 like that.

(Refer Slide Time: 16:05)



Sequential Code

Inclusive Scan

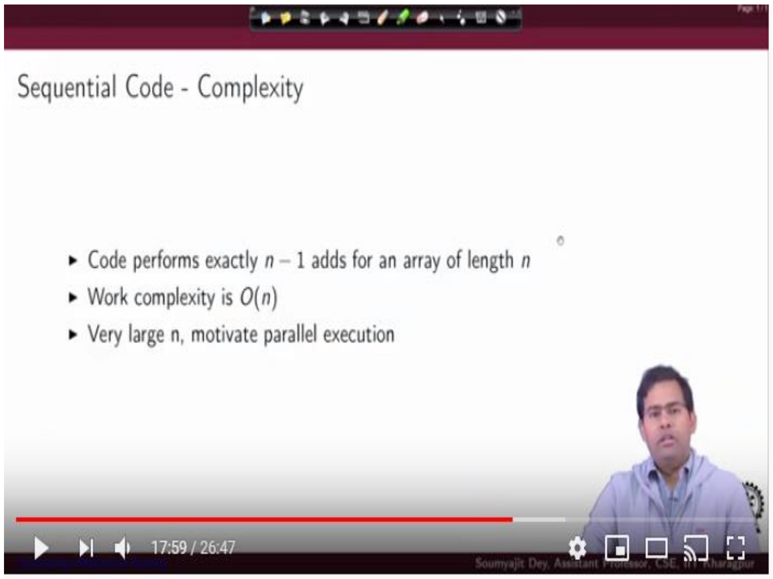
```
void scan( float* output, float* input, int length)
{
        output[0] = input[0]; // since this is a inclusive scan

        for(int j = 1; j < length; ++j)
          output[j] = output[j-1] + input[j];

}
```

So if we just take an example of the code for inclusive scan it is going I mean this is a normal sequence. Shells code of course it is just a simple follow right? Please take a second and have a look. All we are doing is you are accumulating the values in the output array. So in the jth iteration you are computing the output j as simply the previous iterations output plus the current location j value for that input array right input j+ currently accumulated value in output j-1.

This term is necessarily going to give you the output j value so this is nothing but the way you are sequentially iterating over. So that is essentially that code is basically the behaviour like this that one thread of computation either editing like this and providing that output referrals. I hope this correspondence is clear to you. So you know sequential code that is out of program flow will go.

You are just accumulating the value here and you are just a considering the previous accumulated value and so that is let us say in this iteration you are considering the previous accumulated value output arrays location here in the current location of the output array j=1. So this is your output 0 +input 1 right? Then output 1+ input 2 should give you output to output 2 + input 3 should give you output 3 so on so forth.

(Refer Slide Time: 17:44)

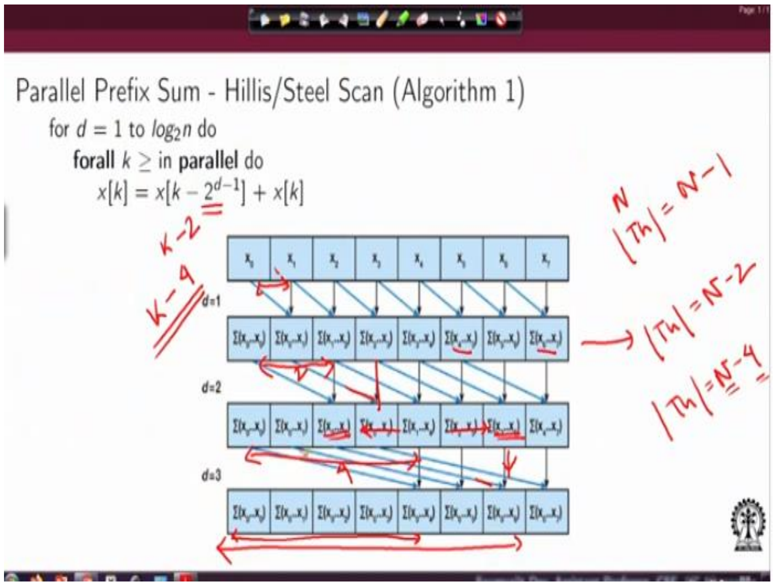


Sequential Code - Complexity

- Code performs exactly $n-1$ adds for an array of length n
- Work complexity is $O(n)$
- Very large n, motivate parallel execution

Now of course this is a it is very easy to understand that these are completely sequential code. So the word complexity is pretty high and for such a simple operation and since there is lot of possibilities for parallel execution.

(Refer Slide Time: 18:03)



Parallel Prefix Sum - Hillis/Steel Scan (Algorithm 1)

```
for $d = 1$ to $log_2 n$ do
    forall $k \geq$ in parallel do
        $x[k] = x[k - \underline{\underline{2^{d-1}}}] + x[k]$
```

We can just do things in a smarter way. So consider initial parallel implementation. So this is Hillis or Steel scan consider this a algorithm here. So what we write here is a parallel version of this follow we are trying to save are the operations we are going to do in parallel. So for every location k right we are saying that okay there will be some parallel additions going on and the number of stages of the parallel additions will be login right?

So we will be doing login number of stages of parallel addition and in each stage what are the better additions we would be doing? It would be that every kth value should be summing up its own value with xk-2 to the power d-1. So d is the current stage value right? As you can see its quite simple its initially d=1. So I am just saying that okay every thread and I should be summing up its location value with the previous value that is all right.

So if d is 1 so then xk-1++xk right? So this is what is going on right? If we just look at the data flow here. So these are initial array for d=1 what we are doing is for every threads I am just summing up its value with the previous locations value right? So of course I would need half the number of threads as is the number of locations.

So this gives me the sums right? Summation these are the partial sums that I am computing. For x1 location I have the summation x0+x1. For x0 location I have the summation x1+x2 like that. So I hope this is clear for let us sum x+ the sum x5+x6 and so on and so forth. And as I go to d =2 let us see what happens. So now in the second stage my stride is increasing right? For the threads I am increasing my stride.

First of all how many threads are now operating? If you see, I have started initially with a 8 sized array and the number of threads I am operating is a basically 1less than then the total number of locations right? So since my total number of locations is 8 I am operating here with 4 threads right? Of course that is required because every thread is going to add its own locations value along with the previous proceeding locations.

Own location means if I am assuming a excess expression we use identity. So just location x for tid right? X has tidth location that is what I would say. So initially I really do not have a half number of threads of the array because every threads is going to add its own location plus the previous location. So if the total is N here the number of threads would be N-1 that are active here. Now what happens in the next iterations as we can see with d =2. And so here we will get k-2 right?

So every location we will start adding up with you know 1 half right? We will start doing a work in stride of 2 right now if that start happenings how is it first of all helping me meet the objective? So again if we just consider then what are the total number of threads that are required here? So first of all that would be N-2 because I have 6 threads operating here. But the issue is why is this a working fine?

So let us inspect that so here I have the value x0+x1 right here I have the value x1+x2. Now if I add this with x0 I will get x0+x1+x2 here right? If I look in here I have x0+x1 here I have x2+x3 operating at a stride of 2. If I add I get x0+x1+x2+x3 so x0 up to x3 right? Similarly, if we just check x4, x5, x6, x7 add them off to x4 to x7 right. Now if we go to the next stage which is d=3 and that should be the last stage here because your N is 8t and log N base 2 would give you d=3 and now you say that okay now what is my stride value?

So since d is 3 so it is per 2. So that means now the threads would operate at a stride of size 4. So you start with a stride of 1 and you are just multiplying your strides right by 2. And of course this should not work. Why? Because first of all let us again write the number of threads you have active.

So here you are working with N-4 threads is just the you are just subtracting the number of strikes the number of threads that are you are basically subtracting the stride value from N because every threads is operating with that stride value. So you can surely having that many strides left as simple as that. So now if you look at the correctness of the algorithm is very easy to figure out because here I have essentially computed the sum up to x0 to x2 right?

Now if we look at a half like this so where do I have the sum from x3 up to sum the next 4 values x3, x4, x5, x6 sitting here right? What is their distance? They are sitting at the half of 4 right? So if I consider these two I get x0 to x6 right? So that is the inclusive prefix sum up to this point. If I consider this point and this point what do I really have here? I am having x1 to x4 I have here just the x0.

So I will just add them up and I would get x0 to x4 right? So I would get an x0 to x4 here. Just take another example and another location. So as we can see that here I have x0 to x1 and look at half of 4 from here. So from here you have x0 to x6 so if you just add them up you have x0 to x2 to x5 and so you will add up and get x0 to x5. You can easily write a formal proof of this it is quite easy to figure out. So essentially we can understand that this algorithm is working right?

We are not going into the formal proof because that would more be a part of algorithm course here but all were trying to introduce here is how the algorithm works. It is easy to figure out illustrated at its neighbourhood values at the shortest distance and then add in the next iteration. It goes to add the next the values on already accumulated in the neighbourhood. And next it goes on increasing its horizon.

And finally every threads will end up the way the accesses are designed the threads will end up computing values from the initial up to each location right? So with that maybe we will end this lecture and in the next lecture we will try to figure out the more refined versions of this algorithm and how this can be implemented and what are the technical issues with this. Thank you.