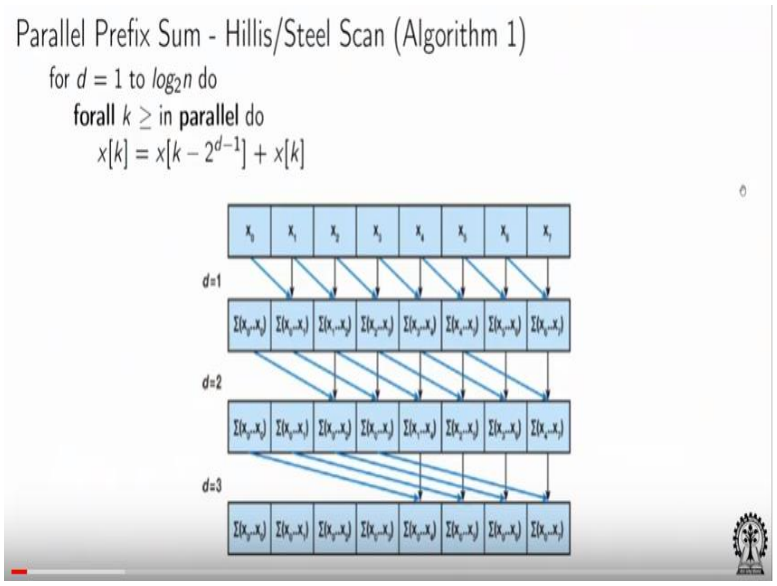
GPU Architectures and Programming

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

Lecture – 34
Optimizing Reduction Kernels(Contd.)

Hi welcome back to the lecture series on GPU architectures and programming.

(Refer Slide Time: 00:30)

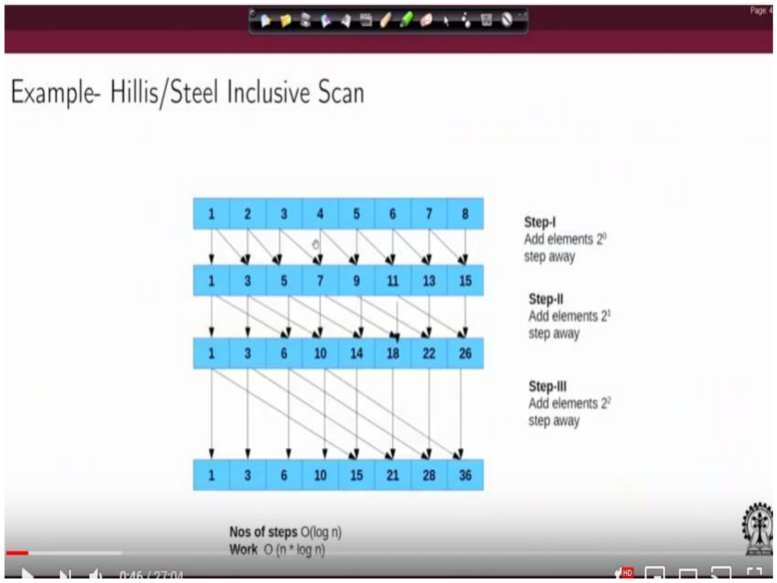


Parallel Prefix Sum - Hillis/Steel Scan (Algorithm 1)

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

So in the last lecture we have been discussing about some algorithm which can perform parallel computation of prefix sum and we argued that why this algorithm would work.

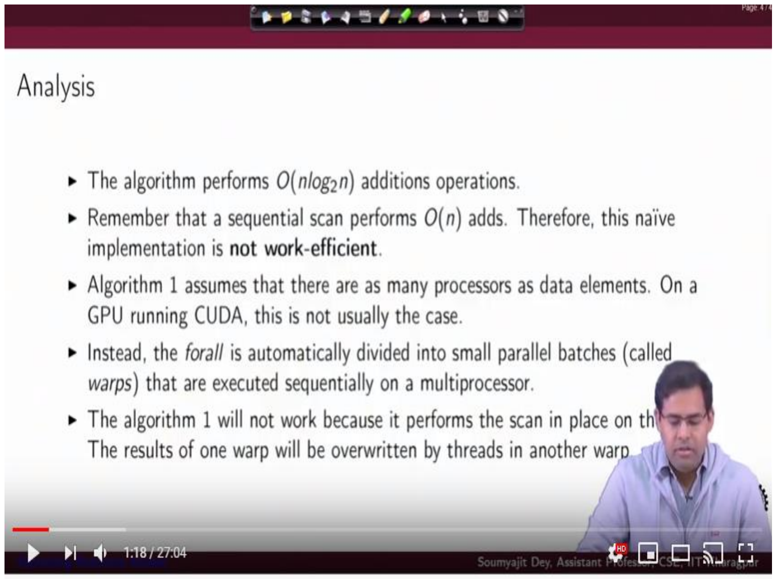
(Refer Slide Time: 00:43)


Example- Hillis/Steel Inclusive Scan

Nos of steps O(log n)
Work O (n * log n)

But let us look at also a picture which kind of just signifies and highlights the properties of the algorithm we discussed that initially in step 1 is adding elements that steps off at a stride of 2 to the power 0, 2 to the power 1, 2 to the power and like that. So in each iteration the number of threads that are active is decreasing from n-1,n-2,n-4,n-8 like that. The number of steps here is the log n of course because that is a depth.

(Refer Slide Time: 01:20)


Analysis

- The algorithm performs $O(nlog_2n)$ additions operations.
- Remember that a sequential scan performs $O(n)$ adds. Therefore, this naïve implementation is **not work-efficient**.
- Algorithm 1 assumes that there are as many processors as data elements. On a GPU running CUDA, this is not usually the case.
- Instead, the *forall* is automatically divided into small parallel batches (called *warps*) that are executed sequentially on a multiprocessor.
- The algorithm 1 will not work because it performs the scan in place on th
  The results of one warp will be overwritten by threads in another warp

And what would we analysis here first problem is that sequential scan performs the order of n adds and this implementation apparently seems to be naive because we also have that many adds there. And if we are but what is the more important point here is that if we are trying to implement these directly in just the way we talked about the loop if we just implement this kind of a loop in form of a CUDA program it is not going to work.

Why? First problem is here we are assuming it seems like were assuming that we have as many compute codes as are the number of active threads. Why would I say that? We are assuming that because you see unless we assume that we are going to have a problem. Why? Because the in case its not so for example in a GPU that is really not going to be so considering a big array you have launched lots of such threads for doing this prefix scan operation and you do not have many threads.

So you are now going to do the processor when you are trying to implement this for a loop in parallel that means its going to I mean how will really you execute this as a CUDA program? The first thing you will do is you do not have such a parallel loop you do have a kernel where inside the kernel you would have only this. So this first of all this is not a representation of the CUDA code. This is just like a way we are trying to tell that how the parallel algorithm would work right?

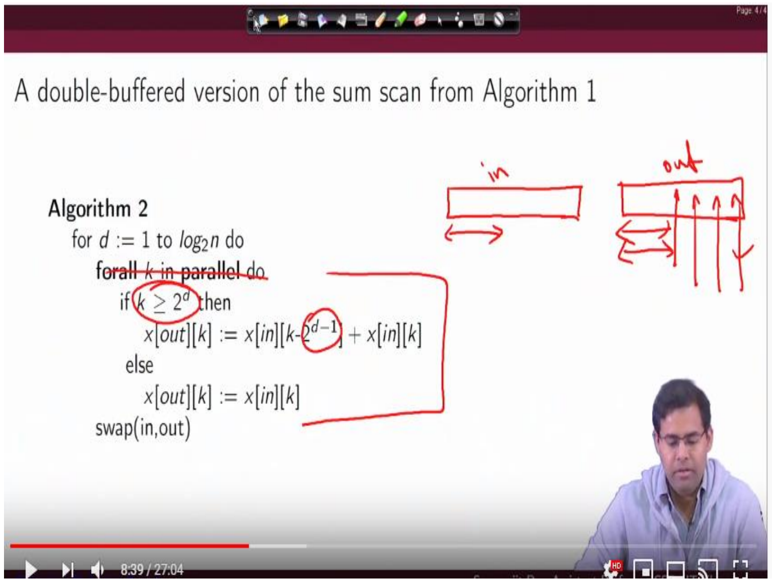
I mean that is something we would like to make clear here. We will say that this is a forall loop inside the forall loop. I am trying to say that this is the activity that is going to happen in parallel right? But when we talk about the corresponding CUDA implementation we do not have something like this. We just define a part thread activity and the part thread activity will be defined that okay each thread will be executing this outer forall loop and this statement right that is how it will work.

So then we will be launching those many threads in parallel and they would be divided into warps. And those warps are at the at the mercy of the hardware scheduler because the warps will be scheduled based on the hardware schedulers algorithm and we do not have any control there to which warps goes faster and which warps get slowed down right. And due to that we will also have problems. What is the problem? The problem is that the results of one warp can get overwritten by the threads of another warp right?

The reason being here the way we have presented that wording things are happening in place. So as you can see that we are writing every thread is writing its results on the same array right? So the moment this computation gives divided across warps there would be locations where you need data you need data across warps right?

So unless we have some synchronization primitive or we have some way to control that how the works interact among each other this naïve implementation. If I just do a paradise and here in a naive way where I mean where every work is warp is working in place things are going to be very bad. So what is the option in that case how can I really make the warps work in such a way that they do not override results computed per parallel partial results computed by other active warps.

(Refer Slide Time: 05:09)



A double-buffered version of the sum scan from Algorithm 1

```
Algorithm 2
  for $d$ := 1 to $log_2 n$ do
    ~~forall $k$ in parallel do~~
      if $k \geq 2^d$ then
        $x[out][k] := x[in][k-2^{d-1}] + x[in][k]$
      else
        $x[out][k] := x[in][k]$
    swap(in,out)
```

One option is that again we are just writing the pseudocode here right? I mean we are trying to give the activity description which you have to implement as a CUDA code. Try it let us first understand that how this scheme will ever that problem is called a double buffered version of the algorithm.

Why double buffer in the algorithm. One example we just showed that in place implementation which we argued that is not going to work in case of work based scheduling of the GPU. If we consider this double buffered implementation we are saying that hold on. Let us consider that we have two such arrays in and out. Updates by the thread will always happen on that out array. So what we will do is first thing we will figure out what is the depth.

So in the CUDA code this thing will need be there right? These all parallel activity these are part thread activity. Every thread falls into the loop in the loop first it has to figure out what is that depth at which does the operating. If the depth is 1 you have n-1 threads working. If the depth is 2 you have n- sorry if the depth is 0 you have n-1 threads working.

If the depth is 1 n-2 threads if the depth is 2 n-4 threads like we have already discussed right? So that decision is taken by this condition right? If so based on that depth I have inactivity at certain parts and threads starting from the rest of the positions are going to work with this much of stride right? So this value is actually deciding what is the stride. As we have already discussed earlier that the stride is equal to I mean n-stride is the number of threads that would be working right but what are the threads going to do?

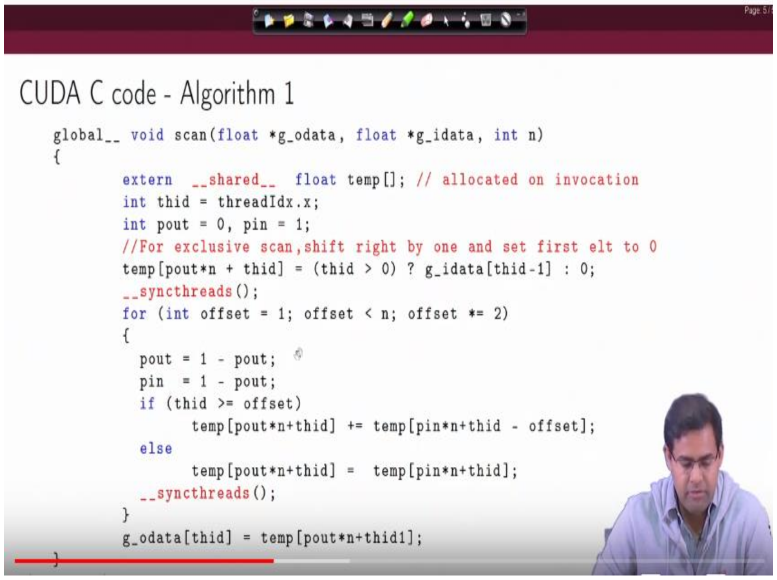
They are going to consider the data values from the in array and do the computations and updates on that out array right? So the threads will do their computation on the out array. If those trails do not belong to the portion where we are doing the computation like if we go back if the threads are here if the threads are from here I mean with 3 ids of here then at some stage they are not going to work right.

I for example this thread will stop working here. Only these threads will remain active. This thread this thread will stop working after d=1 right? So that is the condition which is being signified by this if and then inside this if we are seeing that okay just do the same thing. Just copy data from in and do the updates in the out buffer. If you are not in the working set the thread if then do not do anything but still just completely update things in the out buffer right?

So the thread either computers that means does the addition or it just updates data from the in buffer to the out buffer. At the end of this you do a swap. That means exchange the values from the into the out. That means after this you have a consistent view of whatever this threads has done in the in buffer once this loop is processed by some warp right?

So in that way now the earlier problem can be mitigated and we can have a CUDA implementation.

(Refer Slide Time: 08:44)



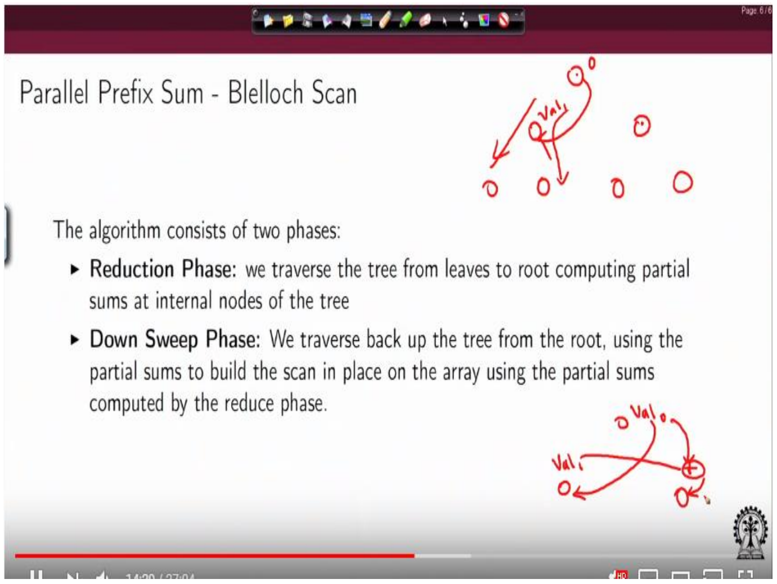
CUDA C code - Algorithm 1

```
global__ void scan(float *g_odata, float *g_idata, int n)
{
        extern  __shared__  float temp[]; // allocated on invocation
        int thid = threadIdx.x;
        int pout = 0, pin = 1;
        //For exclusive scan,shift right by one and set first elt to 0
        temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
        __syncthreads();
        for (int offset = 1; offset < n; offset *= 2)
        {
          pout = 1 - pout;
          pin  = 1 - pout;
          if (thid >= offset)
                temp[pout*n+thid] += temp[pin*n+thid - offset];
          else
                temp[pout*n+thid] =  temp[pin*n+thid];
          __syncthreads();
        }
        g_odata[thid] = temp[pout*n+thid1];
}
```

So this would be the CUDA code here. So as you can see so here this is the loop inside which the above computation we discussed would be done. So the thread idea is first checking whether its before we beyond the offset or not. In that case it will add otherwise it will just simply copy right? We are discussing it at a high level. Every thread would synchronize at the end of the iteration of the loop. That means every thread synchronizes with all the operations inside one depth level and then the entire loop goes to the next level right like that.

Once the entire loop is traversed by the threads in the warp they are just going to copy back from out to in right? And then again somebody some other warps which is progressing. If they require the data they will get the data from the modified in right? So this is how this algorithm is going to work. But yet we can say that this algorithm can be further improved in by using a better technique where we can have to we have to do lesser number of adds.

(Refer Slide Time: 10:03)



Parallel Prefix Sum - Blelloch Scan

The algorithm consists of two phases:

- **Reduction Phase:** we traverse the tree from leaves to root computing partial sums at internal nodes of the tree
- **Down Sweep Phase:** We traverse back up the tree from the root, using the partial sums to build the scan in place on the array using the partial sums computed by the reduce phase.

So let us see that how possible improvement can be done. So this is possible using this Blelloch scan technique here. The idea is to build a balanced binary tree on the input data and you sweep it to and from the root. So you sweep you first to do our normal reduction and then you do what we call as our down sweep. And if we do that essentially after the downstream operation we can get all the prefixes computed.

We will see with an example what we really mean here and it should be easy for us to actually mark out the advantages out of this method. So this algorithm will have two phases. In the first stage we have a normal reduction phase just like a normal reduction computation. If we assume addition just like a normal reduction addition examples we did earlier. So we traverse the tree from leaves to root computing partial sums at internal nodes of the tree.

Now why suddenly have we started talking about the tree? Essentially if we look at a reduction operation and if you want to create a sequence of operations we can say that we will we are first doing computations at stride =1 and then stride =2 like that. And that I can represent in the form of a tree. So let us say this as initial nodes. We are just trying to draw a normal production picture here.

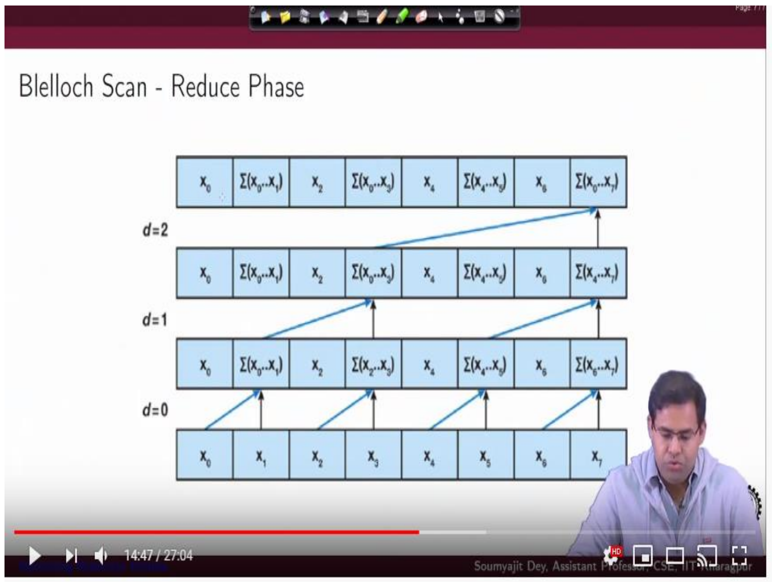
So you do the sums here the local sums we are doing right then you go 1 up effectively this is what we are getting right? If we just try to create a graph here based on how the values are flowing then essentially each thread does a computation at the at stride 1. Then you do a computation like this like that right? So I can say that thread id all the thread ids would progress like this.

This is the initial computation and then the way I am organizing here I am trying to say that okay then these two hellos are going to be added like that. I mean this is one way of looking at it right. But then at the end of the reduction phase what do we really have? Okay let me just redraw it again. So if we create a tree of this computation at the end of the reduction phase we have the overall sum sitting here right?

So after this phase what we will do is we will do a down sweep. That means in terms of the data structure it traverse back from the root to the lower level. And what we do is so what initialise this root with a 0? Suppose I have some value here value 1. So I will copy this with value here. This value comes here and this value 1 plus the existing value here goes to this point right?

So if I draw a better picture I will start with the root being initialized to an identity. If it is an addition then it is 0 we pass it here and whatever the value was here you add it up with this value and put it here. So in general if I said this is value 0 and this is value 1 you put value 0 here you add value 0 and value 1 and put it here. So this is the operation you keep on performing downstream to get the final prefix sum computed and it really works. So let us see how it works in terms of a parallel implementation.

(Refer Slide Time: 14:39)


Blelloch Scan - Reduce Phase

d=2
d=1
d=0
x_0 $\Sigma(x_0..x_1)$ x_2 $\Sigma(x_0..x_3)$ x_4 $\Sigma(x_4..x_5)$ x_6 $\Sigma(x_0..x_7)$
x_0 $\Sigma(x_0..x_1)$ x_2 $\Sigma(x_0..x_3)$ x_4 $\Sigma(x_4..x_5)$ x_6 $\Sigma(x_4..x_7)$
x_0 $\Sigma(x_0..x_1)$ x_2 $\Sigma(x_2..x_3)$ x_4 $\Sigma(x_4..x_5)$ x_6 $\Sigma(x_6..x_7)$
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
14:47 / 27:04
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So a good idea would be to look at the picture here because it would convey the situation clearly. So first of all what we really have at the end of a reduction phase we understand that at the end of a reduction phase at this position we would have the final sum computed of the entire array. Now if we look at the each different phases of reduction so like this is the tree structure I was talking about earlier. So considering this as the lower level notes I can say that here I have computed I have computed in intermediate value right?

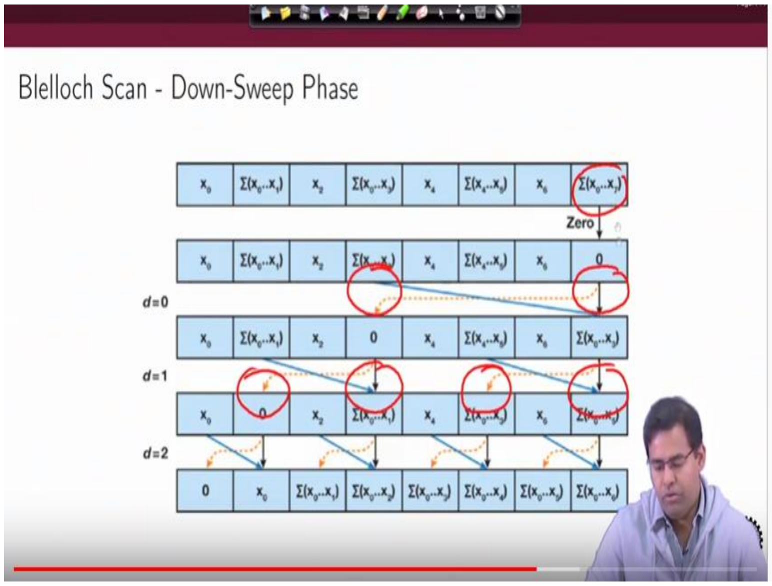
These values are again getting summed up by this thread. Again these are intermediate values. They are getting summed up here and I am creating a tree like this with this being the note root note containing the entire sum right now just try and observe what are the partial sums that we have right now.

So at alternate locations I have no sums but just the value corresponding to that location at alternate locations and other alternate locations what I have is the partial sums right? So here I have summation up to x7 alternate location I have summation up to x5 then in alternate location summation up to x3 summation up to x1.

I have to figure out a smart way such that I am able to compute the prefix sums of course the only prefix sums which is ready with me right now is this. And of course the other alternate locations because this is also prefix sum valid prefix sums are there right? So we have them ready but what is not there is the prefix sums in the alternate locations which I have to figure out efficiently.

Right now the way we will do it is as we explained earlier in this picture that okay we will do a down sweep and perform the operation that I just defined that start with the identity from the root copy that in the left child take the left childs value and add with the roots value and put it in the right child and let us see that why that works.

(Refer Slide Time: 17:07)


Blelloch Scan - Down-Sweep Phase

First let us just observe the figure. So what we will do is here in the down sweep phase we will put the identity here which is 0. And then we pushed back the 0 here okay and then this the input of the down sweep phase first thing we do is we remove this and we will just remember this and push a 0 here in the down sweep phase and put the 0 here.

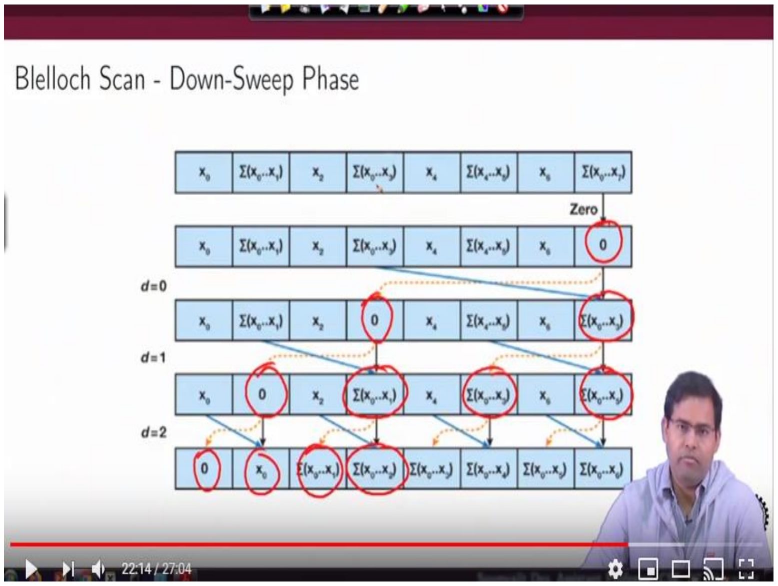
So this is the left child where we copy the value and what was the original content of this location? It was x0 up to x3 so do an addition with this value with x0 to x3 sum. So in that way in the right child you put in the summation x0 to x3 right? Just to repeat this thing in parallel that is the good thing were going to do. So again from this points where we made the modifications in the tree you just keep on doing the same operation.

So you again copy the current locations value which locations the locations which you where you just made the updates the left and right childs you copy it to the left child whatever is the left childs value. The summation x0 to x1 add it up with this value that you just copied put it in the right child right? So that would give you the summation x0 to x1 here and give you a 0 here.

If you propagate it you get the 0 here you and essentially here you are adding 0 with the x1 that was already there right 0 with this would give you x0 here. When you copy from here you put in x0 to x1 and the right childs value that was originally there was x2. So you just add the right cha sorry the left childs value and put it back to the right child. So that is this is what you get.

Again I will just repeat the algorithm. What we are doing were copying the value from the root to the left child.

(Refer Slide Time: 19:10)



Let me just redraw these locations again. So you are propagating it here and what was the original content? This one you are adding you are with this and putting it here. Just follow this you are propagating it here in the left. Whatever was the original content this was the original content of the left right? You add it up with whatever you propagated and put that updated value to the right. Now you see why this works.

So here are propagated x0 to x3 and you are adding up this with the original content which was up to x4 to x6 sorry x4 to x5 and in that way you are getting summation x0 to x5 here right? You are putting in summation x0 to x5 here you adding with original location x6 and here you are getting x0 to x6 again if I look here so whatever you copied here was summation x0 to x3 you copy that to the left child x0 to x3 original content of left child was x4 you add it up to get summation x0 to x4.

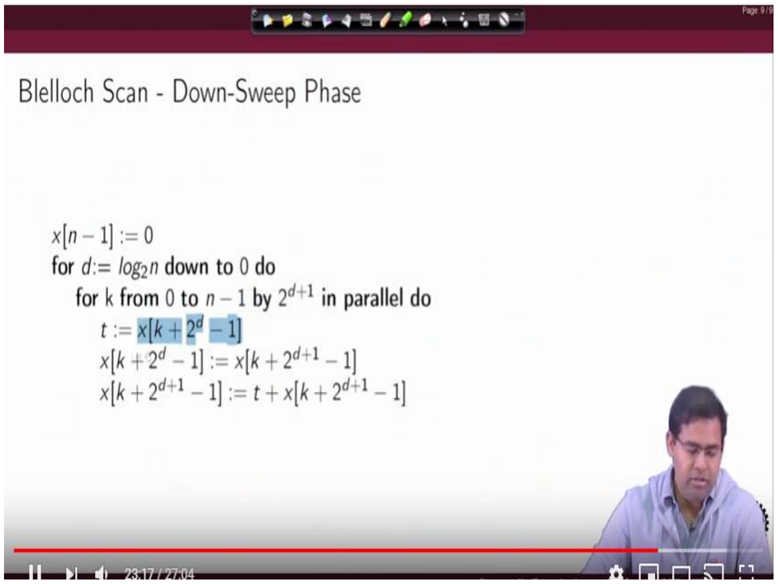
Right? So if we just followed this operation at the end what you get is basically an exclusive scan right? So 0,x0, x0 to x1 like that up to summation x0 to x6 right? So in this way with this parallel sums getting computed you end up the alternate locations prefix sums been computed you already had something here x4 to x5 and thenx0 to x3, x0to x1,x0 to x7 like that. But now you have for every location it done.

So initially after the reduction phase your status was only for the root from the last location or I would say the root location of the tree. You had the total sum done that was the correct prefix sum for that location. But for the other locations there were sums which were actually for certain sub strides sequence of locations and in the alternate locations you just had individual values right?

But by doing this operation you are now able to get when the down sweep phase ends you are now able to get a continuous sequence of prefix sums for the entire array. So I would just repeat the operation that we did once this value is this computation of the reduction phase is done. Reduction phase gives you the prefix sum only for the last location. And for the other locations you have at alternate locations the only the value that was there. And at alternate locations you have some subsequence sums that have been computed right?

So now you start the down sweep phase. So you initialize this last location with a 0 and you start propagating this to each left child and whatever is the content of the left child add it with whatever you propagate and put it in the right child. And in that way when you in the down sweep for the requisite number of depth so log in depth then you end up getting the array with the actual prefix sums.

(Refer Slide Time: 22:50)



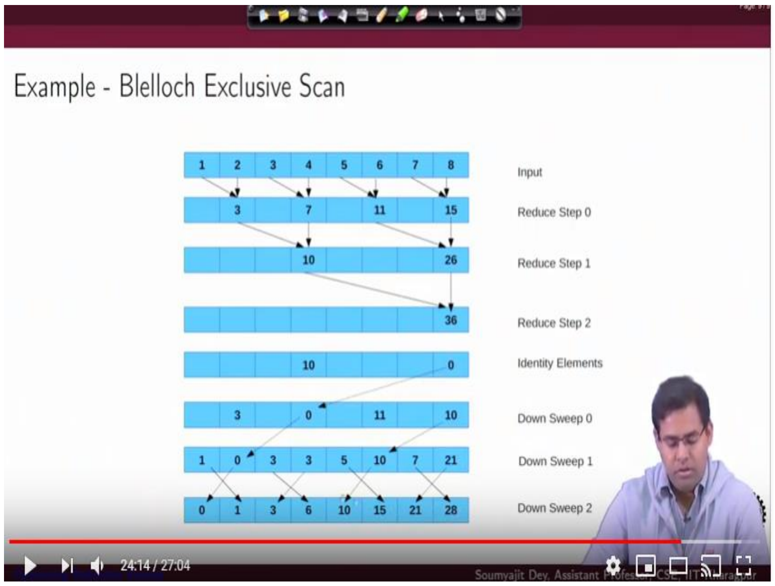
Blelloch Scan - Down-Sweep Phase

```
$x[n-1] := 0$
for $d := \log_2 n$ down to 0 do
    for k from 0 to $n-1$ by $2^{d+1}$ in parallel do
        $t := x[k+2^d-1]$
        $x[k+2^d-1] := x[k+2^{d+1}-1]$
        $x[k+2^{d+1}-1] := t + x[k+2^{d+1}-1]$
```

Okay so if you just look into the parallels pseudo-code here. So essentially you are learning this loop up to log in depth inside each iteration depending on the depth here what you do is essentially you are storing the current locations of value a temporary variable. Then you are propagating this value so you are storing so sorry this is the left childs value which you are storing in the temporary variable.

Then in the left child you are storing the current locations value. And then in the right child you are updating with the current locations value which is this and the previous remembered left childs value the operation that we just discussed and this is going to happen in parallel inside a depth. And then were going to go to the next step.

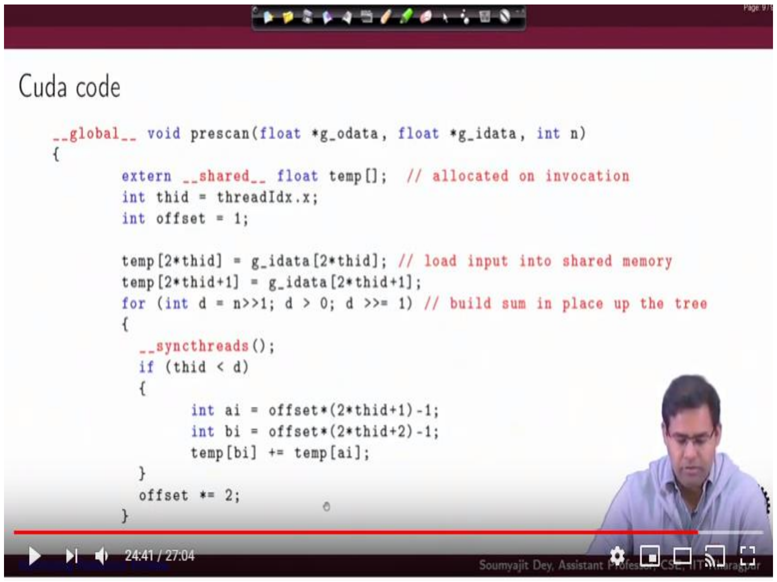
(Refer Slide Time: 23:40)



Example - Blelloch Exclusive Scan

Just look at the picture. So now we have a overall picture of the Blelloch exclusive scan. So as you can see that we have the reduction phase done at the reduction phase then I just have this location containing the inclusive sum value but from the others I just have the either the original values or I just have a subsequent sum value. And then I perform the down sweep starting this location with 0 and actually utilizing all the other location values. And we see that in parallel we are able to compute the value for the prefixes as an exclusive scan at the end.

(Refer Slide Time: 23:40)



Cuda code

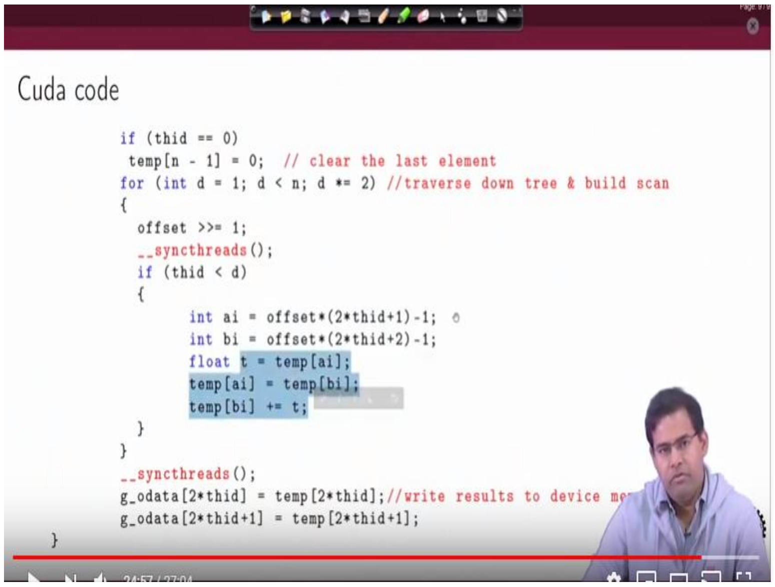
```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
        extern __shared__ float temp[];  // allocated on invocation
        int thid = threadIdx.x;
        int offset = 1;

        temp[2*thid] = g_idata[2*thid]; // load input into shared memory
        temp[2*thid+1] = g_idata[2*thid+1];
        for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
        {
          __syncthreads();
          if (thid < d)
          {
                int ai = offset*(2*thid+1)-1;
                int bi = offset*(2*thid+2)-1;
                temp[bi] += temp[ai];
          }
          offset *= 2;
        }
```

So just a representation of the CUDA code here which would be easy for you to figure out. Given that we have already discussed that what is exactly is going on. So this is the loop where you are performing that so this is a basically the part where you build the sum through the original reductions here right?

So this is the part where you are doing the reduction. I mean we are not getting into the details here. It is kind of repetitive.

(Refer Slide Time: 24:46)

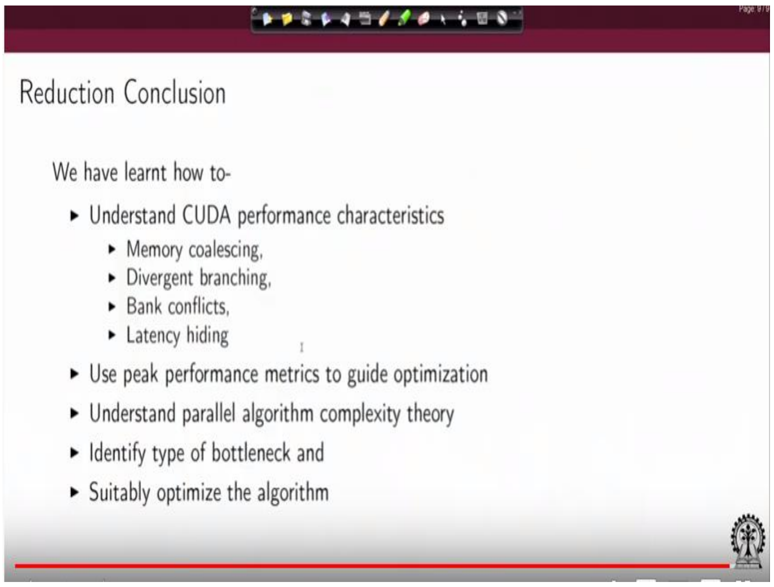


Cuda code

```
if (thid == 0)
 temp[n - 1] = 0;  // clear the last element
for (int d = 1; d < n; d *= 2) //traverse down tree & build scan
{
  offset >>= 1;
  __syncthreads();
  if (thid < d)
  {
        int ai = offset*(2*thid+1)-1;
        int bi = offset*(2*thid+2)-1;
        float t = temp[ai];
        temp[ai] = temp[bi];
        temp[bi] += t;
  }
}
__syncthreads();
g_odata[2*thid] = temp[2*thid];//write results to device me
g_odata[2*thid+1] = temp[2*thid+1];
}
```

And then once the reduction is done this is the next loop where we are performing the down sweep. As you can see this is where we are performing that operation of getting values from the roots to the left child and adding up the left child values with the right child and reaching the right child and at the end doing a sync thread across all the threads in the block. And then finally updating that global memory.

(Refer Slide Time: 25:13)



Reduction Conclusion

We have learnt how to-

- Understand CUDA performance characteristics
  - Memory coalescing,
  - Divergent branching,
  - Bank conflicts,
  - Latency hiding
- Use peak performance metrics to guide optimization
- Understand parallel algorithm complexity theory
- Identify type of bottleneck and
- Suitably optimize the algorithm

So with this we would like to end our discussions on certain parallel deduction algorithms. So what we have we like to end up with the following conclusions that the we try to understand the performance characteristics of these different optimizations. Like how memory coalescing handling divergent branches resolve being brand conflicts and trying to hide the latency by making more threads do their work makes sense. And to create performance awareness programs.

So we explored all these techniques in the context of parallel reduction and then we also started looking into other parallel algorithms as an example and their corresponding CUDA implementations. What we did not really do is explore how I mean what are the performance issues in those algorithms? I will actually I mean actually leave those for people who are interested and maybe some of those will also be part of our assignments.

So these are the issues we have to think right that how to make the other algorithms that we discuss performance our I mean how to create low latency versions of those codes. What are the issues with those codes what are the good and bad things with respect to memory operations latency, cache conflicts or whether they really have some bank conflicts and all that?

So please look those aspects for the other algorithms that we discussed apart from parallel reduction like since in bitonic sort and also the one in prefix sum operations. And with this we would like to end this topic and in the next lecture, we will be starting a new topic here. Thank you for your attention.