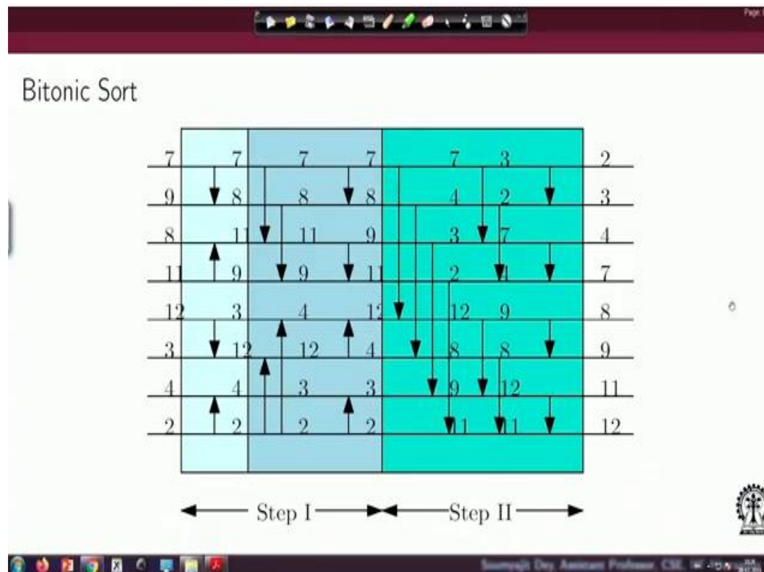
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture – 32
Optimizing Reduction Kernels(Contd.)

Hi so in the last lecture on bitonic sort we have been discussing about the recursive structure.

(Refer Slide Time: 00:30)



So what we identified was that step 1 in the bitonic sort is nothing but application of smaller instances of step 2 and then combining them to create the step 1 right and the next thing was we are trying to figure out what is the step 2? And if you just recall from our further discussions.

(Refer Slide Time: 00:53)

Recursive C Program

```
//Step I
void bitonicSort(int lo, int n, boolean dir){
  if (n>1)
  {
    int m=n/2;
    bitonicSort(lo, m, ASCENDING);
    bitonicSort(lo+m, m, DESCENDING);
    bitonicMerge(lo, n, dir);
  }
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

We figured that the bitonic sort algorithm can be broken into 2 parts. So you apply bitonic sort on the smaller parts right. You break the total input into 2 parts apply bitonic ascending followed by bitonic sort descending on the first part and the second part respectively to generate a bitonic sequence and then apply the bitonic merge which is essentially your definition of step 2 right. So this first 2 steps generate the bitonic sequence then you apply bitonic merge that is the step 2 right.

(Refer Slide Time: 01:21)

Recursive C Program

```
//Comparator
void compare(int i, int j, boolean dir){
  if (dir==(a[i]>a[j]))
    exchange(i, j);
}

//Step II
void bitonicMerge(int lo, int n, boolean dir){
  if (n>1){
    int m=n/2;
    for (int i=lo; i<lo+m; i++)
      compare(i, i+m, dir);
    bitonicMerge(lo, m, dir);
    bitonicMerge(lo+m, m, dir);
  }
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And so the point we are trying to make now is that okay what is step 2 or what is the fundamental behind the bitonic merge process? Now when we went back to our example here we figured that in the bitonic merge essentially what we have is a sequence of comparisons as we

can see. So if the input size is 8 we are applying 4 comparisons here 1 stripes of size 4. So half of the size right.

So this sequence of comparisons are followed by smaller instances of bitonic merge that is the important thing. Now if you look into these parts so we have a sequence of 4 comparisons. Then a set of steps. Now if you look into these parts I can say that the overall thing is a bitonic merge of size 8 and these two are nothing but bitonic merge of size 4 right. So from this recursive structure.

We can say that the merge step can again be written as a sequence of comparisons followed by 2 calls to bitonic merge on a reduced set right. With this understanding if we look into the program here the pseudocode here. So you have this botanic merge with a low and a high and you are given the direction essentially now what will happen is you will make those sequence of comparisons then make the call 2 calls to be precise as we saw for this part and this part you are making 2 calls.

But the calls are going to happen on 2 separate parts on the half sized inputs. So from lo to m = n/2 the high/2 and then for the rest half. What is important is to understand is the merge process? The direction is same right. So in these 3 calls in all the comparisons and in all the merge course the direction remains same. Again I will repeat in all the comparisons and in the merge process the direction remains same. If you look in here that is true right. The direction is same as the desired merge process direction of the outer call.

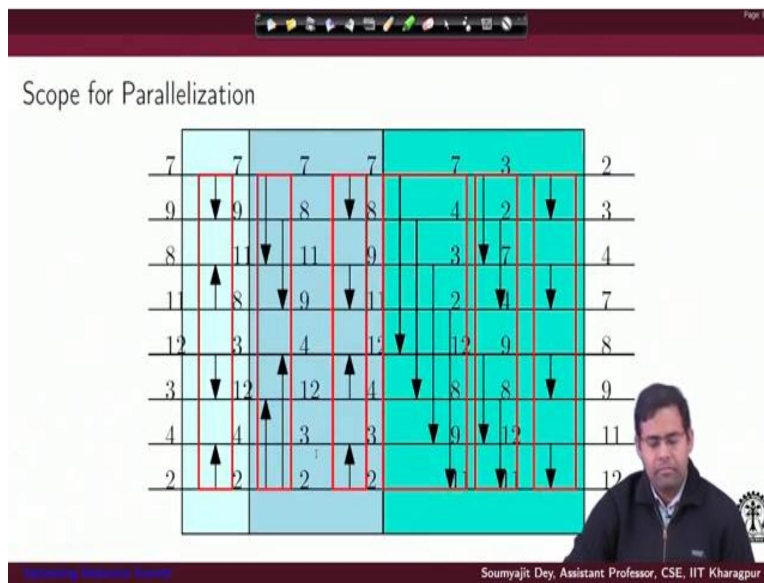
So here the outer call is expecting a bitonic sequences input and is going to provide an ascending output right. So for that we make a call to merge of size 8 with ascending output reduction and here we do the comparisons in that direction and for the sub calls to the smaller merge processes again the direction does not change right. So this is the most important thing to identify with respect to step 2 that I can simply write the merge process recursively like this.

All we need to do we will have first this set of comparisons. The comparisons are happening with a half size stripe. So from the start value you make comparisons up to the midpoint right lo

+ m there is a midpoint. As you can see from the start you are making comparisons 1, 2, 3, 4 up to the midpoint. Each of the comparisons are considering this data with a data at the stripe size which is half of the overall width of the data right.

So you are comparing the ith value with the i + mth value you know this m is the stripe size and the stripe size is again = n/2 so half of the data width as I have been saying right. So I hope this is now clear to you that we have a sequence of comparisons here and this comparisons will have their stride as defined and then after the comparisons we just make calls to consecutive bitonic merge right. So once we execute this overall program we have the bitonic sort done right.

(Refer Slide Time: 05:36)



But now the question is what about the cuda implementation? So first thing we have to figure out is okay let us have a look what are the parallelizations. So as we can see the sorting network provides me the view that with respect to at the same time point what are the operations happening? So these are the operations that are happening in concurrently and here again these are the operations which have no dependency and I can make them happen concurrently right.

So in red we are trying to figure out which are the concurrent operations that we have and as we can see there are lot of concurrencies present inside each stage. So first I have step 1 inside step 1 I have 2 parts right. So first I have this single comparators and then I have this idea of again

applying the step 2 in a larger context and then again applying the bigger merge process here right. So I hope this is clear that why even inside step 1 I am using 2 different colors right.

So the first part essentially tells me applying step 2 in the smallest context and then here you are applying 2 instances of step 2 in a relatively large context and then you are applying step 2 in the bigger possible context here right. So looking at the parallelization as we can see I can define the per thread activity like this right. So I can run these parallel threads right and each thread in this part of the computation of step 1 they can perform this comparisons in parallel.

So I would require 4 threads here half the number of threads as the number of data points. So each thread can perform 2 loads and then do the comparisons put the value in the shared memory right. And then the threads can just do a sync threads here, progress here effectively do this comparisons and then progress and then progress so on and so forth right.

(Refer Slide Time: 07:51)

Formulate Parallel Solution

- Associate every cuda thread block with a sorting subproblem.
- Merge results from each SM to solve the original sorting problem.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So overall we can summarize like this that associate every cuda thread block with a sorting subproblem. So essentially if we have a significantly large sized array to sort you define cuda thread blocks for each of the sub arrays and then you apply the merge on the results for the different thread blocks that you derive. So at the end you have to actually merge results from each of the SMs and solve the original sorting problem.

(Refer Slide Time: 08:25)

Mapping Sorting Subproblem

Thread Block

4 threads for eight elements

Shared Memory Size = 2 * Number of Threads

So fundamentally we need to also identify that what is the size of the thread block? As we discussed that since 4 threads are going to do the job in this case. So if I have an 8 element input I would require half the number of threads okay but what is the shared memory size is going to be same as the input size here for each block right. So it is going to be twice of the number of threads right.

(Refer Slide Time: 08:52)

Comparator

```
__device__ inline void Comparator(uint &keyA, uint &valA,uint &keyB,uint &valB
    ,uint dir){
  uint t;
  if ((keyA > keyB) == dir){
    t = keyA;
    keyA = keyB;
    keyB = t;
    t = valA;
    valA = valB;
    valB = t;
  }
}
```

NVIDIA CUDA SDK Benchmark Suite

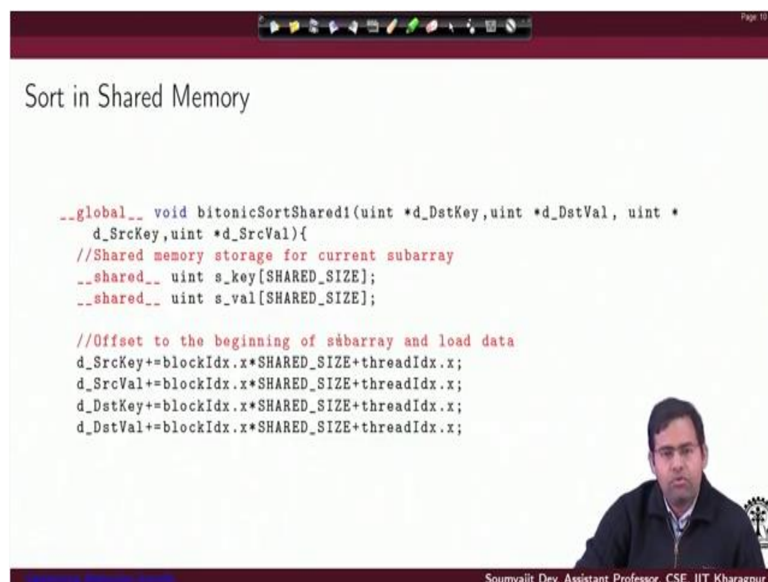
Now these are simple codes that we have taken from NVIDIA CUDA SDK benchmarks right and so first thing we highlight here is the comparators code. So what we have is 2 inputs now we are writing a general situation here. So we have 2 inputs with key value pairs right. We have 2

inputs with key value pairs and we also provide a direction and then for the inputs we compare the key and then we compare it with the direction.

Now as you can see that the idea of this program is similar to the C program comparator that we had. So essentially we are checking it with the direction if i > j and there is a 2 case. Then we do the exchange so we are considering downward arrow sorting networks by this example here and of course if the direction is otherwise then it will represent the upward arrow sorting network right. So as we derived earlier considering dir as 1 I have this, considering dir as 0 I have the reverse right.

So we compared the key and then we compared it which we just compare the result of this comparison I mean 1 or 0 with the direction and accordingly we decide to swap the key and swap the value right. So this is just an extension of the original comparator circuit considering key value pairs.

(Refer Slide Time: 10:41)



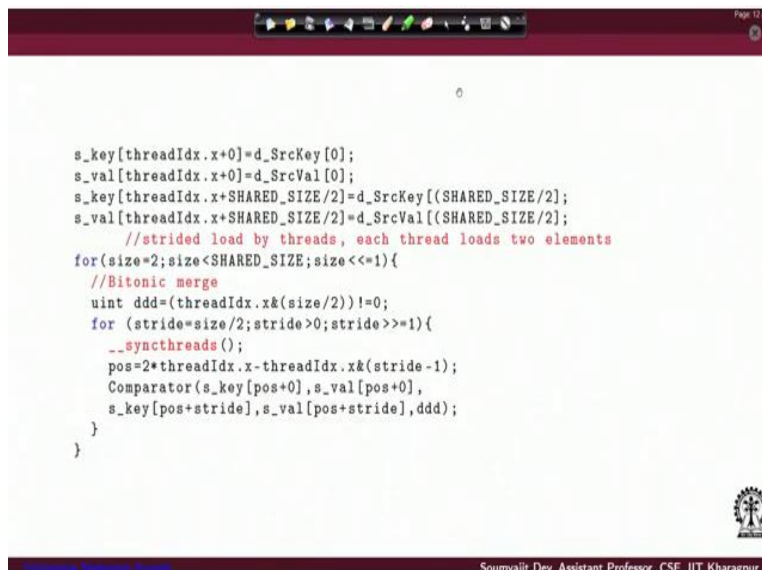
So we are going to do the sorting based on the key values. Now we provide the implementation of bitonic sort on shared memory. So we will have each thread loading 2 values to a shared memory and proceeding through the different stages of the sort. So we define 2 of these shared memory segments one for the storing the key values and one for storing the one for storing the keys and one for storing the values here.

And then we have to figure out what is the per thread activity? That means we need to figure out okay each thread is going to store load and store 2 key value pairs. So they are going to work in which area of the memory. So for the 2 threads the first thing you do is you calculate the offset to the beginning of the subarray that means inside which of these blocks a thread is going to do the load activity right it has to identify that.

So what we do is we check the block ID. So whatever is the block ID you I mean we are considering x dimensional and one dimensional data here so you just multiply the block ID in the x direction with the shared size and then if you do a plus thread Idx dot x you get the offset right that means inside the shared memory corresponding to this thread block what is the offsetted location in the shared memory where this thread is going to load the key, load the value and also the other key and the other value.

We just call them as destination and source key values right. So this part of the code actually performs the computation of that offset. So these are block and you just compute the offsets of the key value pairs right. Because you are considering for each block you have 2 shared memories so every thread is going to load 2 of these keys and 2 of these values and considering only 1 dimensional data here right.

(Refer Slide Time: 13:06)



So once we have figured out that this is the location from which we are considering the threads to load the values into the shared memory. So now the pointers have been shifted to the offsets. So now you can just access the location with this as the base shifted by 0 for the source key and the source value right and you write it back to the shared memories location only with the thread Idx offset.

I hope this is clear as we can remember that each shared memory is being defined in a power block basis right each shared memory segment that we have here is there in a power block basis. So first each thread figured out in the global memory what is the offset or from where it is going to load the data? Is going to load 2 of the data points right and then it figured out using the thread Id based offset into the shared memory that where the locations they are going to write right.

So this is my shared memory from the global memory I have figured out what is the data I will write and then in the shared memory for this block just by using the thread Idx based offset and figuring out per thread blocks. This is the location where I am going to write the key and so this is the shared memory for keys and similarly I have the shared memory for values. So in that I will also figure out where to write the value and so on and so forth.

Now as we see that every thread is going to route 2 loads to the shared memory both for threads and values. So the second load has to happen it has tried of with an offset of shared size/2 right. Because it has first figured out its location the best location where it is going to load using thread Idx inside the shared memory block and then you add that thread Idx with half size of the shared memory block to figure out what is the other location of where also is going to load the key right.

And that it is guessed from this d source key I mean similarly how do I get the load data address from the global memory? Well I have already shifted this to this d source key is now pointing to the location from where you have to just load the first data point right. So that is why you can access it by index 0. So you can just put the index as shared size/2 and that would give me the other location in the global memory from where I am going to load the second data point right.

So just right so this is the second location from where I am going to load the other data point. So first thing we did was we calculated the offsets that means what is the location in the shared memory where I am going to load the data right and then we just loaded the keys and the values at the respective locations with the offsets as provided.

(Refer Slide Time: 17:06)

```
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
  int 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);
  }
}
```

Now we come to the second phase that now I have just figured out that where the loads are going to happen in the shared memory by the thread and each thread is just doing that 2 loads from the global memory to the shared memory. The next thing that is supposed to happen is we are going to enter the step 1 right. We are going to enter into the step 1 in which we are going to build in parallel the bitonic sequence followed by.

In step 2 we are going to figure out how this bitonic sequence is going to be arranged in the completely ascending or descending order right. As we have seen from our earlier algorithm that well all that we will be needing to do is we will first have the sequence in the step 1 and step 2. So here first we show the part for the bitonic merge here that is happening for the different segments that are going on.

Because we have already identified that even for step 1 we just need to carry out the bitonic merge process at the different segments that are there of the program right. Sorry so before carrying out any of this comparisons we need to figure out that what should be that direction of

the comparison and of course what should be the positions where the comparison operation has to be performed right.

So for that we introduce 2 variables here this variable ddd is kind of computing what is the direction? And this variable pos is trying to figure out what is the position, where the comparison has to be performed right. Now if you have a loop into the sequence of operations that we do here right. So let us try and figure out that how to assign this comparisons in as a per thread activity.

Because till now whatever we have done is to figure out how each of these 4 threads are going to load the values right. So the next phase is to figure out how each thread is going to be orchestrating the comparisons that are to be done. So what we do is okay for performing the comparisons.

We let thread 1 do the first comparison right and then we let thread 2 to do the second comparison and then we let thread 3 do the third comparison and thread 4 does the other comparison right or maybe I can just use the actual Ids of the threads sorry which would be 0, 1, 2, 3 right. So these are the per thread activities and then coming to the next part what we do is?

In the next phase of operations so if you may recall this entirely is my step 1 inside step 1 I am essentially executing step 2 first with I mean one-fourth of the input size and then I am executing step 2 again with half of the input size right. So here this is my definition of the per thread activity and then I have this is the part that would be done by thread 0 and then this is what will be done by thread 1 and we will have the following to done by thread 2 and 3 respectively so on and so forth right.

So as you can see what is happening is in the first part we have threads doing the comparisons at a specific location and the next thread is doing the comparison at another location with the direction continuously flipping right. Then when we come to the next part what we have is the threads again doing the comparisons at a bigger stride for 2 threads the directions do not change and again for 2 threads the direction do not change right.

So somehow we need to figure out that how this effect can be brought in right. So for that if you check the way we are actually computing the direction is basically a function of the thread id and the size value right. So we put a for loop here which is trying to figure out that okay in which stage we are operating of step 1 right.

So let me just reproduce the picture here we are just elaborating on step 1 here right and as we see that step 1 has 2 parts. So this is the full picture of step 1. Now since I am working with an 8 size input in step 1 we have already figured out that what we do is here we run step 2 on half the size. So I can just write this is step 2 on half the size here and similarly step 2 on half the size here and then we run step 2 on one-fourth of the size 4 times right.

So essentially I can say the number of times I am running step 2 is going to be controlled by the outer loop and inside each iteration of the outer loop I am doing the number of comparisons which is controlled by the inner loop I hope this is fine. So the number again I will repeat so the number of phases I would say of applying step 2 in different stride size is being controlled by the outer loop.

For example here you are executing step 1 on an input of size 8 right. For that you have to execute step 2 on half the sized input twice and step 4 step 2 on one-fourth the size of input 4 times right. The fact that you are going to use step 2 in 2 different configurations is being reflected by the outer loop right. So the outer loop is controlling in what configuration I am going to apply step 2.

First when I run the outer loop it is setting the size variable at 2 and it is saying that okay you are going to apply step 2 with the comparisons done between 2 consecutive elements okay and that the fact that these comparisons will be done. So essentially you are applying step 2/4 that means you are going to apply step 2 on input of size/4 4 times essentially we will be doing 4 number of comparisons that is going to be controlled by the inner loop.

Look at the next iteration of the outer loop you are increasing size/2. Once you increase size/2 what is happening is? So again I will just repeat once you increase size/2 what is happening is?

You are reconfiguring your way of running step 2 and you are saying that okay now I will not be comparing I mean okay I am going to run step 2 again at configuration where it is input size/2 and that is realized again by the inner loop with suitable choice of size and direction.

So if I just loop that how the values of size and stride are changing so initially when I start well I have size = 2 and stride as 1 and so with that configuration since stride is okay by the way the stride is computed here and that comes out to be 1 because size = 2 and all we are doing is we are just running the inner loop and in each iteration of the inner loop we are actually going to increase this value of stripe right.

So starting from the initial value right so what we have here is we start with sizes 2. So when we figure out that okay in that case stride would be 1. Now since stride would be 1 i would get in here and here I am going to execute the comparison operation by all the 4 threads right. So I will execute 4 different comparisons right. So they are all done in 1 shot by the 4 threads in parallel and then I am trying to decrease the size right.

So that would mean since the integer variable size stride 2 was already 2 and stride sorry stride was already 1 if I decrease by having it with 2 it goes to 0 and the loop does not execute anymore right. So in that way I have this 1 iteration of the loop and inside this 1 iteration of the loop I have all the threads executing 1 comparison in that way I get 2 of the 4 of the comparisons done in parallel and the comparisons are done on adjacent values which is dictated by the stride variable and the size value was set to be 2 right.

So now if the next iteration when I come back for the outer loop I would have size as increased by this factor of 2 right because of this left shift and but still it is smaller than the shared size so I come to the other part of the step 1. And here I am now going to execute these 2 different parts of step 2, 1 for have the input size 1 here and the other here. Let us look at how this is done getting realized.

So now I have said the size value as 4 right. Since size value has been said as 4 I am now going to execute with stride at 2 because stride is size/2 and then again the loop will again execute with

a stride value of 1 and then the stride will go to 0 and it wont execute anymore right. So when I execute with size as 4 and stride as 2, let us see what has happened okay. Again I will get into this loop and all the threads will execute comparison.

But now the comparisons are going to happen with the stride of 2 right that is why you get these 4 comparisons happening at a larger stride size I mean part on my picture and maybe we look at the original picture. So I am talking about these 4 comparisons here right. As you can see they are happening with a stride size of 2 yeah. And then in the next iteration of this loop well I will again enter into the next iteration.

Because I will half the stride let it become 1 which is still greater than 0. So now I will again perform 4 of this comparisons on address and data points right and their directions would be like this. So this loop ran with for 1 iteration in the first size essentially it executed 4 different comparisons. Essentially all of them were the atomic instances of step 2 running on one-fourth the size of input that is the step 2/4 that I am trying to say.

So essentially and that covers all the 8 inputs right in 4 pairs. Here again I am running it I am just repeating. So step 2/2 sized that means step 2 on the input size of 4 that is right 2 of the instances and the way it is being realized is that well we set the size variable again now as 4 and with the size variable as 4 we get 2 possible stride values that is why this inner loop is now going to iterate twice.

In the first iteration is going to perform 4 comparisons, in the next iteration again 4 comparisons, in the next iteration in the first iteration the comparisons are at strides of 2, in the next iteration the comparisons are going to happen at strides of 1 right. So in that way this code is actually achieving the overall idea of performing the step stage 1 or step 1 of the bitonic sort. Now observe the different ways the directions have changed.

So as you can see here the direction was we have repeated I am just repeating this direction was continuously switching. In the next stage the direction remains same for the first 2 threads and then switched for the next 2 threads and that pattern continued here. To realize this switching

activity of the direction we do a bitwise end based realization of this value using thread Id and size.

If you can check these values there you will see that how this expression is actually able to I mean reproduce this idea of direction switch where when I said the size as 2 the direction switches very frequently. When I said the size as 4 the direction switches up for 2 threads I mean the first 2 threads have the same direction and the next 2 threads have the other direction. That is how the direction switch activity happens.

And then for the position again I have this expression which actually takes care of identifying for each thread Id what is the starting position? Right. Because for thread Id 1 this has to be the starting position for thread Id this followed by there is a switch in the starting position yeah so this is also something interesting just the comparison positions for each of the threads the use for thread Id 0 right.

For the next thread the starting positions actually change. These are the starting positions for the next thread and for the other threads for the thread with Id 2 again it is this and for the other thread is this right. So these are the different starting positions that we have. Now to achieve this sequence of positions as a function of the thread Ids we use this access expression right. So I mean you will need to check how this actually reflects those switches of position by the thread Ids.

(Refer Slide Time: 35:13)

Bitonic Sequence Creation

size = 2 size = 4

stride = 1 stride = 2 stride = 1

$ddd = (1\&(4/2)! = 0) = 1$

$pos = 2 * 1 - 1\&(2-1) = 1$

$pos + stride = 1 + 1 = 2$

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now this is also an example through which you can identify how everything is going on. So here we have actually noted down in the as a part thread activity that thread Id 0 is performing, this operations thread Id 1 is performing, this comparisons thread Id 2 is performing, this comparisons thread Id 3 is again performing this comparison so on and so forth. And we have also done some example computations of triple d position, position + stride etc.

(Refer Slide Time: 36:03)

```
//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];
}
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Yes now we come to the other part of the code where we are kind of trying to figure out what is the is the step 2 here right. So if you see in the step 2 you have this uint which is basically the direction right. Now in step 2 first of all the important thing is we already have figured this out

that the direction does not really change right. So you compute that direction and it just remains the same right.

So and then the thing that keeps on changing is at what stride value you keep on performing the comparisons right. So maybe we will take this up in the next lecture and with this we will end the current lecture. Thank you.