

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


**Lecture – 28**  
**Optimising Reduction Kernels**

Hi, so welcome back to the lectures on GPU architectures and programming so, in the last weeks we have been discussing more on memory access and (()) (00:35) issues and throughout the last few lectures while studying about warps and their scheduling, the divergence and how memory access is (()) (00:44), we have fundamentally understood the GPU architectures memory hierarchy and how it can be used to write optimised programs.

**(Refer Slide Time: 00:56)**

Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
<b>Optimizing Reduction Kernels</b>	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6



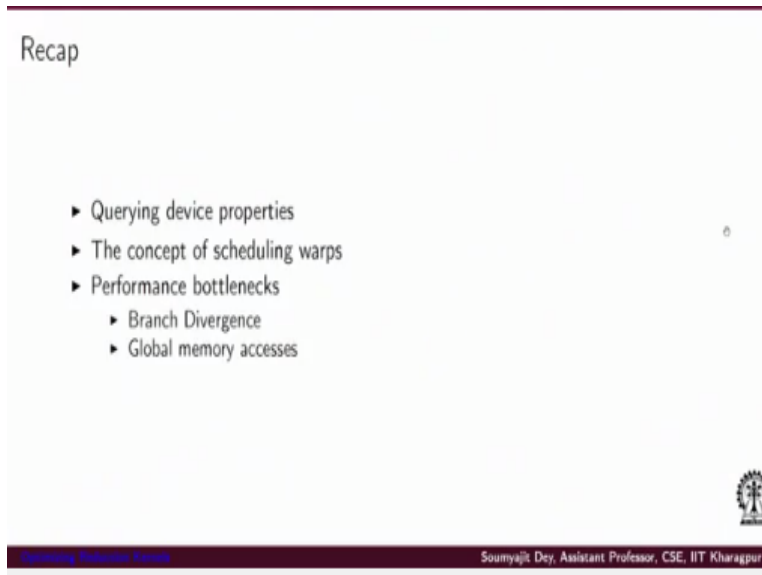
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

**(Refer Slide Time: 01:00)**



So, with this background, we are going to this topic of optimising reduction kernels that means, we will try to use this concept for some regular programming jobs, the primary one being reduction kernel.

**(Refer Slide Time: 01:07)**



So, these are the concepts which we have already covered and we will be trying to use those concepts here.

**(Refer Slide Time: 01:14)**



Now, before getting into reduction kernels, we have to understand that in general, while doing parallel programming what is important is to understand the concept of parallel patterns. Now, a parallel pattern would mean specific pattern of tasks that is a specific execution order and data access order of tasks which can be found occurring very frequently in some algorithm. So, if I have a specific sequence of computation and a specific sequence of data access that is occurring very, very frequently that is what we call as a pattern, if that has lot of parallel jobs embedded in the pattern in terms of computation or communication of data.

It is the parallel pattern and we can see such parallel patterns occurring in very large number of computation intensive things we do, for example, matrix multiplication, for example convolution and also the reduction operations.

**(Refer Slide Time: 02:25)**

---

## Reduction Algorithm

- ▶ Reduce vector to a single value via an associative operator
- ▶ Example: sum, min, max, average, AND, OR etc.
- ▶ Visits every element in the array
- ▶ Large arrays motivate parallel execution of the reduction
- ▶ Not compute bound but memory bound



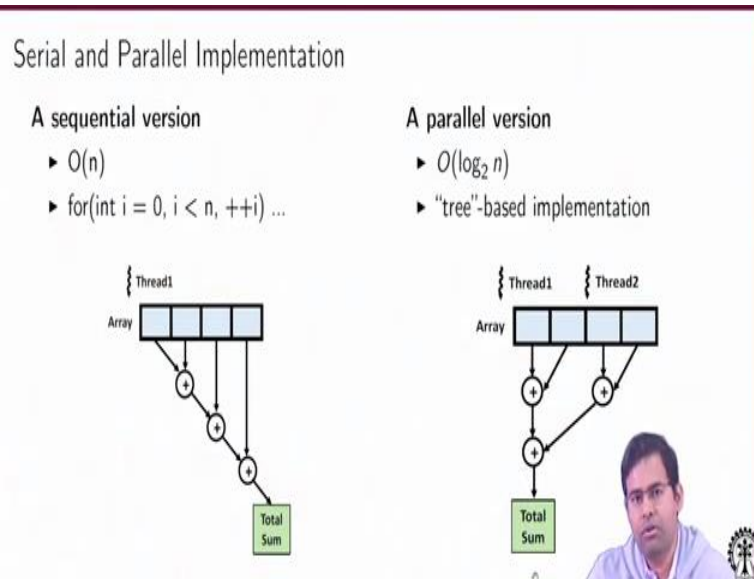
So, we will try and focus first on reduction operations, now typically what is the reduction operation? So, a reduction algorithm or reduction operation is primarily, the name of; it is the generic name for any operation that takes as input of vector and reduces it to a single value via some associative operation. For example, sum, min, max, as you can see that these are all associative operations.

So, like suppose, I am computing the min, it is same as if I write; right so, these are generic name for operations which reduce a vector to a single value and for fundamental, the operation is the associative operation and these are operations which may visit large number of elements in the array, I mean so, you have lot of elements and you carry on these associative operation to get to the final value and fundamentally, we will need to visit every element in the array and perform the operation.

Now, since and this interesting from the point of view when I have a very large array to work with, so a very large vector, lot of values and my final goal is to reduce it to a single value. Now, why is this interesting; because this is kind of a pathological work load, so in this case as you can see that there is not much of compute operations to do but significant amount of memory operations to do.

And since, there is lot of memory operations involved, we feel that all most of the optimisations we have discussed earlier should have a role to play in case of reduction algorithms and that is what makes them interesting in this context.

**(Refer Slide Time: 04:31)**



So, we just start with the parallel sum some implementation, right so, you have a lot of numbers and you to sum them, if you have a sequential version, it is going to an order of  $n$  operation, if you are going to do a parallel version which we have seen earlier in some other examples earlier is going to be a  $\log n$  operation, right because you have threads which are working on different parts of a large array, the array is in sized and you are going to do the sum.

And then, you are going to compute the sum of sums so on and so forth, so it is going to be a  $\log n$  operation.

**(Refer Slide Time: 05:07)**

## Parallel Reduction Algorithm

To process very large arrays:

- ▶ Multiple thread blocks required
- ▶ Each block reduces a portion of the array
- ▶ Need to communicate partial results between blocks
- ▶ Need global synchronization

Problem:

- ▶ CUDA does not support global synchronization

Solution:

- ▶ Kernel decomposition



Now, if I am going to do a large; this kind of parallel reduction in large array, then first thing is I will require multiple thread blocks, right because one thread; one block can only accommodate 1024 threads, I need more than that, so definitely multiple thread blocks. Now, each block can reduce a portion of the array, now once that has been done, I need the results to be communicated, right.

Because each block may reduce a portion of the array and these portions will need to be reduce further but that would definitely need global synchronisation across blocks. Now, we know that using sync thread kind of mechanism, we can only synchronise inside blocks and CUDA does not support global synchronisation, so essentially you have to design a reduction kernel for a block and you have to call it multiple times and use the results of each block to do the computation over the results in a significant number of iterations.

**(Refer Slide Time: 06:15)**

## Kernel Decomposition

- ▶ Decompose computation into multiple kernel invocations
- ▶ Kernel launch serves as a global synchronization point
- ▶ Negligible HW overhead, low SW overhead

Figure from 'Optimizing Parallel Reduction in CUDA' by Mark Harris

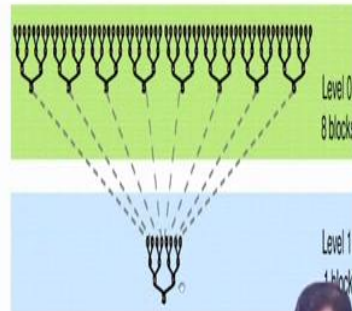


Figure: Multiple Kernel Invocations

So, our purpose here, we need to understand that this is just a small picture we are trying to show, if you have a very large size data, it needs to be decomposed into subparts so, you decompose the computation of the overall sum across multiple kernel invocations and the kernel launch serves as a global synchronisation point that so, after every kernel launch, when you get the data back, those are the synchronisation points from which you can gather the data.

And again, launch further kernels, right, so this is kind of sequence tree of kernel launches which we are trying to show that you have large data, with each launch you get some part reduced, and finally it comes down to 1 block.

**(Refer Slide Time: 07:01)**

## Optimization In Reduction

- ▶ Metrics for GPU performance:
  - ▶ GFLOP/s for compute-bound kernels
    - ▶ One billion floating-point operations per second
  - ▶ Bandwidth for memory-bound kernels
    - ▶ Rate at which data can be read from or stored into memory by a processor
- ▶ Reduction has very low arithmetic intensity
  - ▶ Take 1 flop per element loaded
- ▶ Strive for peak bandwidth



So, here we have examples of codes of how to do these reduction but we will actually, use them more for the assignment purpose and for our discussions, we will focus more on a single block reduction and how that can be accelerated. So, before getting into that let us understand what are the matrix for GPU performance which we will use of course, this is something that we have seen earlier also, this is just a mere recap.

So, ideally we will like to achieve more number of gigaflops per second; flops is floating point operations per second, right, more number of gigaflops and so, this is basically about the compute the total I can do, for compute bound kernels but in this case, it is more of a memory bound kernel, so I will like to see that I am being able to use the full bandwidth of the memory that is the full rate at which data can be read or written into the memory sub system, whether I have able to use that.

Now, in this case, if I am just doing a small reduction operation, it has low arithmetic intensity as we have seen other; right, earlier also right, so it is more like we are going to optimise for achieving the highest possible memory band width that I can get here.

**(Refer Slide Time: 08:22)**

### 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

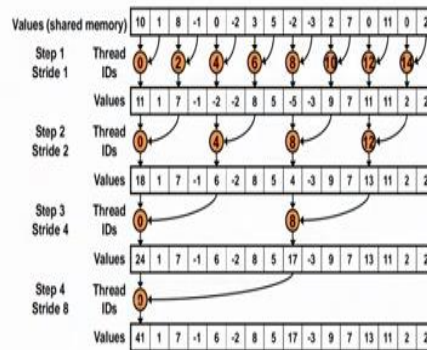


Figure: Reduction with Interleaved Addressing and Divergent Branch



So, the first possible technique here for reducing a block here would be to use interleaved addressing. Now, this is some algorithm, which you have studied earlier also, it is just a brief recap. So, what we do is; we have this many data points to add and what we are doing is you



know, we have launched half the number of threads, right I mean, essentially there are threads and but we are not utilising those threads here.

There must have been threads which have all the threads have copied the data to the shared memory by the way, I will just repeat these are optimisation when you are talking about it, we are doing it in the shared memory, assuming that threads have done a part thread data load into the shared memory, right. So, this is the first job, each thread loads one element from global memory to shared memory.

And then, I will actually engage half of the threads to perform the first level of addition, a thread adds 2 elements and half of the threads are actually deactivated after the initial load because they do not have anything to do because one thread can add 2 elements. So, all the threads together collaboratively load data, then half of the threads perform the step 1 of addition. This addition is performed in a stride of 1 because you just use the threads; thread IDs corresponding memory index value and the next value, right.

And then, in the next iteration, you just increase the stride so, the thread ID; the thread will add the location; its own locations value plus the data sitting at a stride of 2 and then stride of 4 and then stride of 8, in that way finally, the 0th thread will compute the final sum, right.

**(Refer Slide Time: 10:13)**

---

### 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];
}
```



So, this is the reduction kernel, again I will repeat, we have seen it earlier but here we have to repeat it because we need to understand the starting point and what are its flaws, right. So, first thing you do is; you load the data into the shared memory right, so this is the load operation and then once that is done after a sync thread, then all the loads are done, you start doing a reduction. So, this is a reduction loop, you start with the stride of 1 and in each iteration of the loop, you reduced by one level.

So, stride of 1 to the first level reduction, second level; you operate with the stride of 2, so essentially you operate on consecutive data points that is stride of 1, you operate on 2 stride valued data points, so that stride of 2, so that is what we have here. So,  $s$  equal to 1, you do 1 level, then you multiply by 2, you go to stride of 2 and then you again do the addition for if, this on this values that are loaded in  $s$  data shared memory array and their index by the  $tid$ , right.

So, each step does the addition at a stride of 2, then in the next iteration, at a stride of 4 because you again multiply by 2, now what are the good things and the bad things here; where you have parallelism, you have the threads loading the data in parallel and then you have the threads doing the addition in parallel although, half of the threads are not used. The bad thing is the modulo arithmetic done here is very slow.

So, for every thread, you are doing a thread ID percentile to  $s$ , right, so you to identify that whether this thread is the one which is supposed to do the job addition, were whether this time is going to see  $tid$  because as you can see with every level, half of the threads further go idle, whatever were the active threads in the earlier iteration, half of them would go in active, right.

So, in every iteration, you take the  $tid$  and do a percentile  $2s$  to find out whether the  $tid$  will be active or not. If it is active, then you do an addition with the stride of value  $s$ , right and you keep on doing this until and unless, your this  $s$  is up to the half of the block dimension that means, you have reach the; you have actually summed up all the values and you are on the; at the last level and then at the last level, you have the value available in  $s$  data 0 which is a final sum.

**(Refer Slide Time: 12:53)**

## Reduction 1: Host

- ▶ The GPU kernel calculates data per block
- ▶ Partial sums computed by individual blocks
- ▶ Results will be stored in the first block elements of the global memory
- ▶ Final addition need to be done on this reduced data set
- ▶ By launching the same kernel again

And that is what is written back into the global memory, so this was our reduction kernel, for this kernel, you can write a host program and of course, it will be quite complicated in case, you are going to launch it again and again for reducing in different parts of the memory.

**(Refer Slide Time: 13:06)**

## Reduction 1: Host Code for Multiple Kernel Launch

```
...//cudaMemcpyHostToDevice...
int threadsPerBlock = 64;
int old_blocks, blocks = (N / threadsPerBlock) / 2;
blocks = (blocks == 0) ? 1 : blocks;
old_blocks = blocks;
while (blocks > 0) // call compute kernel
{
    sum<<<blocks, threadsPerBlock>>>(devPtrA);
    old_blocks = blocks;
    blocks = (blocks / threadsPerBlock) / 2;
};
if (blocks == 0 && old_blocks != 1) // final kernel call, if still need
sum<<<1, old_blocks / 2>>>(devPtrA);
...//cudaMemcpyDeviceToHost...
```

**(Refer Slide Time: 13:11)**

## 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



So, these are host code, which were not discussing right now, we will actually provide some nice assignments using them and they are those will be useful and it will be described at that level but try and understand the kernels execution here. So, this is the reduction kernel, right and as you can see, first problem is it is highly divergent, why because inside the loop, you have this percentile operation which the thread may or may not satisfy and the modulo arithmetic which is computing this is very slow.

Warps are very inefficient due to the reason that in every warp, you have divergence and anyway half of the threads do nothing after performing the global load and also the loop is expensive. So, some statistics here, considering an array of this size, if you perform this; if you actually execute this reduction kernel, multiple times using multiple kernel launches using a GPU tesla K40, this is the bandwidth and execution time that we can see here, right.

So, just to keep that this is not an execution of 1 kernel launch, we are actually following this picture of orchestrating multiple kernel launches and finally, getting the value, well inside each kernel launch, the code will execute is this one. In each launch, you actually reduce some part of the array. So, as we understand that there are significant number of problems in this reduction code, first of all it is highly divergent and warps are inefficient.

**(Refer Slide Time: 14:52)**

## Reduction 2: Interleaved Addressing

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

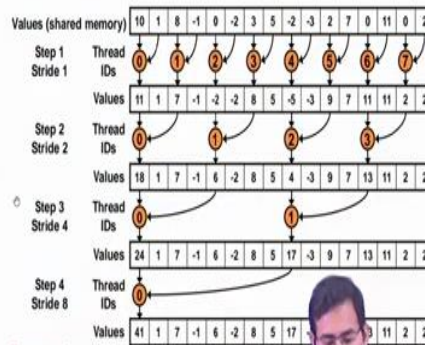


Figure: Interleaved Addressing Replacing Divergent Branch



Soumyajit Dey, Ass

The percent operation also is very slow and we will like to replace it so, what can be the next possibility? The next possibility is that you do interleaved addressing that means, you replace the; first thing is I will like to remove the divergent branch from the inner loop, right and also I will like to remove this divergence but still of course, I will use strided index that will help me to remove this percentile operation.

And also, the situation that in every execution of the warp, consecutive; half of the thread IDs, I mean, actually diverge, right. Now, how can we do that? If we look back into the pictures here, let us understand the divergence here. So, in the first iteration itself, we can see the divergence because these all started inside a single warp, inside a single warp at this point due to that percentile 2s, tid percent and 2s operation, the divergence came in.

Because only this thread progressed, right so, essentially inside the warp, you are getting half of the work done, right. Now, this does not loop very bad here but consider large number of warps working for a big block of code, right, for a big thread block, so that would mean, inside each warp, you have half of the thread is not doing the addition at this level and again, half of the threads not doing anything here.

Try and understand there is a difference between the situation that you have half of the threads going inactive and half of the threads inside a warp going inactive. So, again I will just repeat;

there is a difference between half of the threads going inactive, while there is a property of the operation here but if half of the threads inside a warp going active, then your warps are not efficient, whatever operation the other threads are supposed to do, they are not synchronised inside the warp.

Alternative would have been that whichever were the active threads, whichever were the threads that are supposed to do the real addition, if I can pack them inside the same warp that would be more efficient because as you can see here; here, inside the warp, half of them are progressing, the reason is the thread IDs are what; 0, 2, 4, 6 like that, have the thread IDs of this thread itself being 0, 1, 2, 3 like that.

Then, I looking into a warp of size 32, I could have said that okay, all the threads are working, so originally there would have been a higher number of threads which have done the load but then, inside the warp, every thread is working, so try and understand the difference, half of the threads getting inactive at each level is one thing but still if the threads which are working on having consecutive ID that would mean that they are packed inside does not work.

And when the warps execute, I have fast progress right, now that is what we achieve with this reduction 2 here, so we will do some modification in the code to ensure that the although, half of the threads going inactive but the threads which are really working are of consecutive ID that would ensure they are executing inside a warp and they are all progressing in parallel because if you extend this picture, you have 32 threads which are inside a warp.

That would mean in one lock step, this step of the reduction would get done, which will not be the case if you extend this picture for the earlier reduction 1 kernel, you can understand this. Now, so with the differences you can see between these 2 pictures is with respect to the thread IDs that which thread is doing the job. So, here I still have half number of total threads doing the job at each level but the threads are all sequential in index, they form a common war.

I have removed the divergence, so whoever has the job to do, they will be doing the job maximally, there is no idle slot; that is not the case that a warp is divergent and some of the

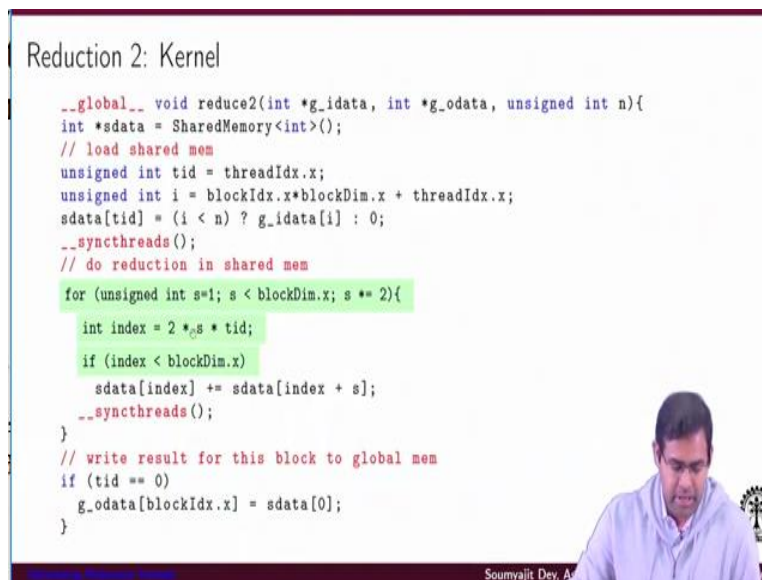
threads are not working, at least in the initial parts of the reduction now, that would significantly speed up the execution of the code, when I launch the kernel multiple times and reduce in different parts of the program.

So, how to do this? So, essentially what I want is still my stride is going to increase from 1, 2, 4 like that but the thread IDs is going to be like this, right. So, while doing the reduction in the shared memory, this is just a snapshot of the reduction in 1 kernel, these were the problems. The first problem was this tid percentile operation and that was slow and that was also getting the divergence.

**(Refer Slide Time: 19:49)**

```
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];
}
```



But now, you do as the different thing so, I have a more complex access expression, earlier my access expression was identity map right, so every tid will map its corresponding index data in s data but now, what I do is; for every tid, I multiply it with s and 2 to get the index where it is suppose, to warp. So, initially s is 1 that means, if the tid is 0, it is working on the 0th location, if the tid is 1, it is working on the second location, if the tid is 2, it is working on the 4th level so on and so forth.

So, that is what we want here, right, tid1; these are the thread IDs, it is working on location 2, tid 2 working on location 4, tid 3 working on location 6 like that and that is achieved by this modified access expression and how long will this work; as long as the index is less than the

block dimension and of course, the stride remains the same so, the striding mechanism remains the same from 1 to multiplied by 2, that is a hop; in every hop you multiply by 2, you start from s equal to 1.

But you change the access expression from my simple identity map; 2, 2s multiplied by tid and that gives you this nice access pattern of threads. So, with this modification, you have warps getting utilised, lack of divergence, so you remove the divergence of the threads and you remove the complex percentile 2 operation but this also will suffer from the shared memory bank conflict issue which we will see soon.

**(Refer Slide Time: 21:34)**

Reduction 2: Analysis


Interleaved addressing with divergent branching

Problems:

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

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



Source: Soumyajit Dey, Asst. Prof.

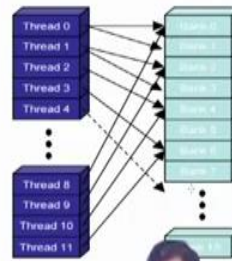
First, let us understand why we will have the shared memory bank conflict, so as we can see that we will have a shared memory bank conflict here, now if you look into this picture so, every thread is accessing 2 consecutive locations right, the next thread is again accessing 2 consequent locations so on and so forth, right.

**(Refer Slide Time: 21:56)**



## 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



Souryajit Dey, Asst. Prof.

So, with this, if you look at the way the threads access the shared memory, so thread 0 access bank 0, bank 1, thread 1 access bank 1 and bank 2, like this if so, I draw the picture for a collection of threads, I would see that I will have inside a warp, I will have 2 threads accessing the same bank in parallel, right. Since, every thread access as parallel banks, the number of threads in a warp is equal to the number of banks.

So, I have a 2 way conflict for every thread, the accesses are not uniform across banks, every thread is accessing 2 consecutive banks so, I have a bank conflict here, right. So, if more than 1 thread attempts to access the same bank, we know, we get the bank conflict and that needs to be resolved now and the way that you can resolve it is you have to again perform some transformation into the code.

Now, again we will try to see that so, every thread, this access is doing, they are accessing into consecutive banks, right.

**(Refer Slide Time: 23:11)**

### 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

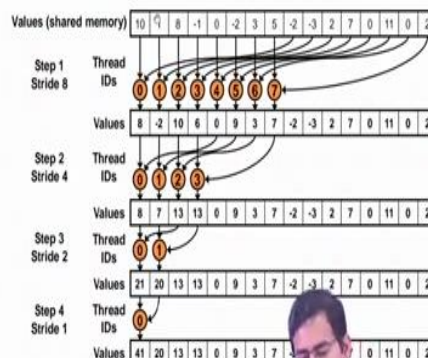


Figure: Reduction with Sequential

So, how can this be modified? It can be modified by doing a sequential addressing, so you do not go from stride value equal to 1 and keep on increasing the stride but rather than that, you keep on decreasing the stride. Now, does this help? Because then, when the threads are going to perform the loads, the loads will not have; loads from the shared memory you will be able to remove the conflicts.

So, this is again something I think before getting into this, you should go back and read once the literature that we studied about shared memory so, at this point we will be stopping today's lecture and we will resume in the next lecture from this reduction 3 and so, just to summarise in the 2 reductions that we have discussed. The first was the original reduction where, let me just through the picture, we had half of the threads working.

But the way the thread IDs are being arranged inside every warp, I have half of the threads working that was a bad thing, in order to remove that, what we did was, we came up with nice and intelligent access expressions, so that inside every warp, I have all the consecutive thread ID is working that gave me some speed up but as we can see if I do a memory access analysis, there is an issue of bank conflict here.

And we need to remove that, we will see how that can be remove by doing further modifications to the expressions in the program and with this, we would like to end our lecture here, thank you.