Parallel reduction as warp unrolling:

In this video, we are going to further increase our parallel reduction algorithms performance using concept called warp unrolling. In parallel reduction with interleaved pair approach We used this code segment to perform accumulation of 1 thread block iteratively. So we set offset value to the half of the block size in the first iteration and then we keep dividing it by half in each iteration, until our offset value reach zero. Let' assume we are launching our kernel with 128 as thread block size, and if we do not perform multiple block unrolling technique we learn in the previous video, so we have to launch the grid with number of threads equal to the number of elements in the input array. So let's try to visualize how this code segment accumulate elements in a single thread block. Since our thread block size is 128, Data block which is going to accumulate by a single thread block has also 128 elements. For the first iteration offset value is set to half of the block size value which is 64 in this case. Now let me divide our data block in to 4, 32 data chunks, so we can clearly visualize what happens here. In the first iteration threads with threadId less than offset value 64, will be the ones performing the accumulation. So only first two warp will perform accumulation in first iteration. First warp will accumulate elements from first and third 32 data chunks and store the results back to first 32 elements in the array. Second warp in this thread block will accumulate second and fourth 32 elements data chunks in the array and store the results back to second 32 indices in our array. In this iteration only 2 warps perform any accumulation. Both 3rd and 4th warps did not perform any accumulation. But if we consider a single warp, with in that warp all the threads follows same execution path, So there is not warp divergence in this iteration. For second iteration, our offset value is 32. So in this iteration, first 2, 32 elements data chunks are going to accumulate by first warp. Other 3 warps will not perform any accumulation in this iteration. But with in a warp, all the threads follow same execution path, so there is no warp divergence in this iteration as well. In the third iteration however, our offset value is 16, so only first 16 threads in the first warp is going to perform the accumulation. So in this iteration, we can see warp divergence in first warp. But remember other three warps does not perform any accumulation, so those warps does not have divergence code to execute. In the fourth iteration, our offset value will be 8, so only 8 threads will perform the accumulation, so in this iteration also, we can see the warp divergence in first warp. Next three iteration with offset values 4, 2, and 1, are also going to have warp divergence in first warp in the same way as, third and fourth iterations. So by implementing reduction using interleaved pair approach we got rid of most of the warp divergence shown by the neighbored pairs approach, but still for last five iteration in accumulation loop, first warp will exhibits warp divergence. Now remember here only first warp exhibits warp divergence. But in our naïve neighbored pair implementation, all 4 warps shown the warp divergence in most of the iterations. We can avoid the warp divergence show by the first warp, using technique called warp unrolling. So let me quickly show you how to do this. The code we are going to use here, is very similar to interleaved paired kernel we implement before, So let me copy it and paste it here. Now let me change this code to avoid warp divergence. Ok, This is what I'm going to do. I am going to unroll the last five iterations of this for loop step by step. So we have to stop this iteration when we reach offset value 32. Now when this loop ends, still the work load done by the final five iteration remain to be done. But we need to perform those steps with first 16 threads only. So to isolate first 16 threads, we have to add a condition here. But if we perform condition check in this way, to separate first 16 threads, that will introduce warp divergence to first warp again. So here is what we are going to do. Instead of unrolling final five iterations I am going to unroll final six ideation in this for loop. So here, we are iterating in this loop until our offset value is 64, Not 32. Now we have to perform final 6 iterations, starting with the offset value 32. So only first 32 threads has to perform the accumulation. So let me add the condition checks to separate first 32 threads. Now this condition check will not induce any warp divergence. Then I am going to have pointer with volatile modifier here. Volatile modifier guarantees that memory load and store to the global memory happen directly without using any caches. Then let me perform the manual accumulation we previously done by the for loop. First we have to perform the accumulation of elements with offset 32. After this step, all the elements in the thread block has been sum up to the elements correspond to the first warp. Then we can perform summation of elements with offset 16. After this step, all the elements in the thread block is sum up to first 16 element in the thread block. And in this way we can perform the summation with offset value 8, 4 ,2 and 1. Notice that all the threads in the first warp is going to perform these summation with given offsets. But steps by step we are summing up the whole array into the elements at the left side of this array. For example when offset value is 8, then first 16 threads in the array has summation of all the elements in the original array, so we have to only take those elements to the account. But still all the threads in the first warp is going to perform the summations. Ok this is it for implementation, Now if we run this example, you can see the validity of our implementation.