Parallel reduction as warp divergence example:

As I already explained to you, if the threads with in a warp executes in a different path then there is a warp divergence in the kernel. . And our device will arrange all these paths together and execute the code sequentially which will result in heavy performance penalties. Unfortunately our previous parallel reduction implementation also has warp divergence. For example, in this diagram you can see 128 threads belong to a single thread block and each slot in this table belongs to a one thread and I have wrote down the thread ids for each slot as well. In our previous implementation, in the first iteration only threads with even number as thread id perform any effective work. So only 50% of threads perform any work And these threads spread across all four warps of this thread block as well. In the second iteration only threads with thread id which is multiplier of value 4 perform any summation. So only quarter of threads perform the summation and those threads also spread across all the warps as well. So in each iteration other than last couple of ones, we have warp divergence in every warp in the thread block. In final two iterations, only couple of warps will show warps divergence since the threads perform sum is spread across only couple of warps. So our previous reduction implementation has lots of divergence code. Luckily there are more than one way of avoiding warp divergence in this type of kernels. Here, we are going to implements reduction algorithm in two new ways to avoid warp divergence. As the first solution we will make neighboring threads to perform effective work in each iteration. In the second method we are going to use interleaved pair approach as a solution for divergence in reduction algorithm. Let's look at each of these approaches now. In the first approach we are going to make sure that all these neighboring thread are the ones performing this summation. For example in previous neighboring pair approach when we perform algorithm on 8 element block, in the first iteration only threads with even number as thread id perform the summation. Threads with odd value as thread id did not used to add the values. However in our new approach, we make sure that neighboring threads perform the summation. So in the corresponding first iteration first four consecutive threads will perform the summation. Ok let's see an example of this approach. Let's say we have a data block with 8 elements. And in the first iteration, first four threads going to perform the summation. So T0 thread will add first two elements, and store the results back to the first index, and T1 thread will add next 2 elements and store the results back to the second index and so on. Then when it come to next iteration still consecutive threads will be the one's execute the summation. In this case T0 and T1. In our previous implementation in the second iteration T0 and T4 where the ones execute the summation in second iteration. And in the third iteration only T0 will be the one execute the summation. If we use 128 as data block size so our thread block size will also be 128 So in the first iteration first 64 threads or first two warps will perform the summation and second two warps will not do anything. But still there is no warp divergence since with in a warp all the threads follow the same path. In the next iteration Only first 32 threads, or first warp will perform the summation and all other three warps will not do anything. In the 3rd iteration however, only first 16 threads or half of the first warp will be performing the summation so from this iteration onwards, there will be warp divergence. But notice, here we have one warp with divergence per iteration from this iteration onwards. But if you remember, in our initial neighbored pair approach when we consider 128 size data block all four warps had warp divergence until last two iteration. Ok let's see the implementation now. The main function for this implementation is almost similar to what we have in the previous neighbored pairs implementation. So I will not go through that. So let's look at our kernel now. In the kernel. Let's set local thread id value and then calculate global thread Id value first. Then we have to set local memory pointer with given offset for the corresponding thread block, so that we can access global memory using this local pointer for this thread block. Then we can have our boundary check now. . And then we are going to perform usual iteration while multiplying offset by two in each iteration. Now we need consecutive threads to perform summation. For that we calculate index value for each thread in the block based on the thread id and offset value. We can use this condition check to limit the threads which are going to perform the summation. And we need all the threads in the block to finish executing one iteration before any of threads in that block move on to the next one so here, we are going to have syncthreads() function call as well. After all the iteration ends first element in the thread block will have the partial sum for this thread block so we have to store it to the our partial sum array. Now let's run this implementation and check the validity. Ok, in the output you can see that it printed out GPU and CPU results are same. So our implementation is a valid one. Ok, Let's move on to the next way of solving warp divergence using interleaved pair approach. In this approach also we are forcing the summation of elements to happen in consecutive threads. And we are going to start the offset of the elements which are going to added together in a iteration from reverse order compared to the previous approaches. For example, in the first iteration we will set our offset value to half of the block size. So if we consider data block with 8 elements, offset value will be 4, hence in the first iteration first thread will accumulate first element and fifth element in the data block. Second thread will accumulate second and sixth elements in the data block and so on. In the second iteration we will divide offset by half. So for second iteration, offset value will be 2 hence first thread will accumulate first and third element, and second thread will accumulate second and fourth element in the current data block. Notice here, we are performing in-place reduction so output of one iteration will be the input to the next iteration so first element for the second iteration contains the summation of first and fifth element in the original array, and third element for second iteration have third and seventh element in the original array. and after second iteration first element in the array now have summation of first third fifth and seventh element in the original array. In the last iteration our offset will be one and when the offset reached one we stop iterating and first thread will accumulate first and second element in the current data block. Now after this step, first element in our array now contains the summation of all 8 elements in our array. Ok, let's see the implementation now. This is almost similar to what we done in previous implementation. The difference is that we initialized offset value to half of the block size and then we keep dividing it by 2 in each iteration. And in the condition check, since we need consecutive threads to execute the accumulation steps we check whether the thread id is less than the offset. So in the first iteration only first half of the thread block will perform the summation. In the next iteration only first quarter of thread in the block perform the summation and so on. And here also we need all the threads in one thread block to execute one iteration before moving in to next iteration henceforth I have used syncthread() function here as well. Then we have to store the summation to partial sum array as well. Now if you run t