Parallel Reduction with Loop unrolling:

In this video we will look at important concept call unrolling in CUDA programming. Unrolling in CUDA means verity of things. However goal of any kind of unrolling is to improve the performance by reducing instruction overhead and creating more independent instructions to schedule. Let's first look at simple unrolling concept called loop unrolling. In loop unrolling rather than writing the body of a loop once and using a loop to execute repeatedly, the body is written in the code multiple times. By doing so we can reduce or entirely remove the enclosing loops iterations. and by reducing or remove iterations of an enclosing loop, we reduce loop maintenance instruction like condition checks in the loop and number of dependent instructions as one iteration have to perform after another. we call number of copies we made in the loop body as loop unrolling factor. For example, here we have loop we used to accumulate an array in CPU. If the array have 100 elements, then we need to iterate 100 times with this implementation. However if we replace this loop from the following in loop then with the loop body we accumulate 2 elements ith element and I+1 th element, so the loop will iterate for half of the original size, which in this case 50. And remember, this kind of loop unrolling is most effective at improving performance for sequential array processing loops, where the number of iterations are known prior to execution of the loop. Unrolling we are going to apply here is somewhat different than our loop unrolling example in this sequence code. You can refer unrolling we are going to perform here as thread block unrolling. But the main purpose, reduction of instruction overhead remain same. In the all previous implementations, we create threads equal to number of elements in the arrays. But if we can reduce the number of thread blocks we need to perform our parallel reduction. Now here is what we are going to do. Before the partial\_sum loop in the interleaved pairs kernel we are going to manually sum up two data blocks to one block. So the amount of thread need to reduction will be reduce by half. With this simple manual addition, we can reduce device workload almost by half. So after we made this change first and second data blocks will be sum up together by first thread block, and next data blocks will be sum up together by second thread block and so on. Ok, let's implement this approach now. Keep in mind all these reduction implementations done by assuming that we are going to launch one dimensional grid. Let me name our kernel as reduction\_unrolling\_blocks2. And it has same argument list as our previous kernels. To access each single element in each data block I need local tid value here. So let me assign threadIdx.x to a variable called tid. Now this step is optional. As you can access this directly using threadIdx.x variable. But I like to use "tid" variable. So that's about it. Since we are unrolling two consecutive data blocks using one thread block each of our thread blocks have to access data block twice the distance as the thread block id. So to calculate element index for each thread based on the data block location we need data block offset based on the thread block id. So let me first calculate data block offset here. Now we can simply add tid value to block offset to get global data element index. And then let me calculate local data pointer, so that our threads can access elements for this thread block using that pointer as well. Now we can unroll two data blocks. So here if the index is with in the size of our array, we can sum up two elements from two consecutive data blocks. . Now before doing anything else, we have to make sure that all of the threads in our thread block finished thread block unrolling part. We can achieve that by using syncthread() function call here. Then we can perform partial accumulations step for our unrolled data block in iterative manner. So here I'm going to use interleaved pair approach to some up data blocks, So let me copy that cord segment from our previous interleaved pair implementation. And that's it for our kernel. The difference here is that we how manually unrolled two data blocks at the beginning of our code. Now what if we want to unroll 4 data blocks instead of 2. Actually the implementation is almost similar to this one. All you have to do is change in the block offset value here to calculate offset for four blocks, so here instead of multiplying by 2 we will multiply it by 4. So let me quickly add the kernel for that unrolling as well. Let me copy the unrolled two blocks kernel and paste it here, and change the name to reduction unrolling blocks 4. Then we can change the block offset variable in this way. Then in unrolling statement, we have to perform 4 unrollings. So here, our boundary check will be index + another 3 blocks, because we are going to unroll 4 blocks, so we have to check those indices are with in the size of our array. And the we can assign values from each consecutive data blocks for variables and finally add those together. And that's it for 4 blocks unrolling kernel. If you want to unroll 8 thread blocks, all you have to do is to change the block offset variable and preform unrolling for 8 elements. Then you will have the 8 block unrolled kernel. Now when you launch these kernels in main function, make sure that you launch the kernel with correct amount of thread blocks. For example, for 2 block unrolling kernel, we should have half of thread blocks than the previous implementations. So you have to divide the grid size in X dimension by two. And for 4 block unrolled kernel, you should have quarter of the threads than usual grid and so on.