Performance comparison of reduction kernels:

In this video we are going to look at another possible way of improving performance of reduction algorithm implementation, and then we will compare the execution time of all the reduction implementations we done so far, and finally summarize the reduction implementation. Here is our completely unrolled reduction kernel. We can further increase the performance of this kernel by using template parameters. Now these condition checks are perform at runtime. So the introduction of these condition checks is adding kind of overhead. If we can some how make our compiler to avoid compiling these code segments with condition checks which are going to fail at runtime based on the thread block size, then that will increase our kernels performance as well. With template parameters we are going to do just that. So here we have the implementation with template parameters. Here, I have added template parameter called iblock\_size and in each of our manual block unrolling condition checks we are going to check whether this parameter is greater than the particular value. So at the compile time these condition checks will be validated against template parameters and remove those condition checks which are failing, and this will result in minimum number of instruction to execute in the kernel. Now from the main function we have to set these template parameters. So in our main function, I have added switch statements, which will launch our kernel with correct block size values as template parameters. And we can validate the outcome of this implementation by running this program now. Ok, as you can see from this output, you can see that our new implementations output and CPU output is same so this implementation is also a valid one. Ok, let me summarize what we done so far with reduction implementation. Our first reduction implementation was naïve neighbored pair implementation. Then we identify the warp divergence of that kernel, and then we improve the performance by making consecutive threads to perfrom summation in improved neighbored pairs implementation. Then we further increase our reduction implementation using interleaved pair approach. This approach got rid of warp divergence in all warps other than the first warp and also in this implementation we had proper memory access pattern as well, and you will learn about this memory access pattern in upcoming section. Then we further improved interleave pair implementation by unrolling data blocks and reducing the GPU work load. Then we got rid of warp divergence in first warp using warp unrolling technique. And finally we further improve our implementations performance by completely getting rid of for loop in the reduction kernel. So these are the steps we followed so far. Now let's compare the performance of these kernels. In your project under the section 2 folder, you can find a file called, 13\_reduction\_performance\_comparison.cu. In this file, I have launched all above mention kernels one after another and I have include timing code to measure the execution time for all these kernels. Now when measuring the execution time, you have to take the memory transferring to account as well. So before memory transferring I have recorded current CPU clock cycle and then we have to record finished clock cycle after we transfer the reduction kernel's results to the CPU and we have to record finished clock cycles only after we calculate the final GPU results in the CPU. So at this point, we even transfer the reduction kernels results to CPU as well. In this way I have done this time measurement for all above mention kernels one after another. Then I have run this program and record the execution times. So let me show you a chart I created using those data. This chart highlights the execution time of each of our reduction kernels. First bar in this graph corresponds to CPU execution time. Now notice our first implementations like naïve neighbored pair implementation and interleaved pair implementation took even more time than CPU implementation. So as you can see, if we do not design our GPU implementation properly, then the performance will be worst than the CPU implementations. But well designed GPU implementation can achieve way better performance than CPU implementation. As you can see from the last bar in this chart, which corresponds to completely unrolling kernel, took only half of the execution time as CPU implementation. Now iss this the best we can do. The answer is no. We can achieve way better performance than this. After you learn shared memory and shared memory access patterns in upcoming sections, you will be able to improve the performance of our reduction kernel even further. But with only the knowledge of execution model, This is the best we can achieve for now.