Parallel reduction as synchronization:

In any parallel programming paradigm synchronization between threads is vital aspect of programming. Here, we are working with more than 1 thread, and often we need to order the operations perform in these threads. So, in this video, we are going to discuss about couple of synchronization functions in CUDA, cudaDeviceSynchronize and syncthreads() functions. cudaDeviceSynchronize function provides global synchronization between host and device. In CUDA, often asynchronous calls like kernel launches are made from the host. Using cudaDeviceSynchronize function We can block the host application execution until all the device operations like kernel execution are finished. And if you look at the programmes we execute so far, you can see the synchronisation in almost every program. On the other hand, syncthreads() function provides the synchronization with in a block. And this function should call from only device code. And it will force threads with a block to wait until all the threads in that particular block to reach that particular point in code. And the global and shared memory access made by all the threads in the block prior to the synchronization point or syncthread() function call will be visible to other threads in the block after the synchronization point. And remember this synchronization happened between threads with in a block. For example, if we launch a kernel with thread block having 128 thread so there will be 4 warps executing the instruction in this kernel. In this diagram, each of those warps represented by an arrow. As we discussed earlier, CUDA follows SIMT execution paradigm. With in a wrap, all the 32 threads execute same instruction. But between warps there are no such guarantees. So let's say warp 2 starts execution first. This execution will be independent from other warps. But if our kernel has syncthreads() function in the code, then warp 2 has to wait in that statement until all other warps reach that point. So all the threads in the block will reach that synchronize point and then only any warps for this block can execute further instructions. Ok let's see a usage of syncthreads() function with parallel reduction algorithm implementation. General problem of performing commulative and associative operations across a vector is known as the reduction problem. In our case we are going to implement accumulation of array of integers as parallel reduction algorithm. Sequential implementation of this algorithm is quite easy. You just have to iterate through the array while adding each element to a global variable. Even though this is straightforward operation, if we have thousands of mega bytes of array, then parallel implementation would outperform the sequential implementation. But remember, usually when we are converting this in to parallel code, it will be much complex than straight forward sequential implementation. Addition is commutative operation. So we can sum the array in any order. So here is what we are going to do. We are going to first partition the input vector in to smaller data chunks and each data chunk will be sums up separately and finally, we will add these partial sums to get total sum. This diagram demonstrate steps we are going to follow exactly. In the implementation we will use thread block size as our data chunk size. So each thread is responsible for adding up given data chunk. After thread block finish its accumulation The result will be stored in to separate array. Each slot in this new array will populate with partial sum of each thread block. So the size of this new array, would be the number of thread blocks in our grid, if we use 1D grid. Which means the value of the grid size in X dimension is our this new array size. After we calculate partial sum of all the thread blocks then we can transfer this partial sum list to host side to do final summation, or if this partial sum array is big enough, then we can again perform reduction in GPU for that array as well. . Now let's see how to sum up particular data chunk. There are multiple approaches to sum up a data chuck. We will look at few of those in upcoming videos. But here we are going to look at the approach called neighboring pairs accumulation. In this neighboring pairs approach we are going to calculate sum of a block in iterative manner, and in each iteration selected elements are paired with their neighbor from given offset. for the first iteration We are going to set offset value as 1 and in each iteration this offset value will be multiplied by two. And number of threads which are going to do any effective work will be divided by this offset value in each iteration. Let's say, we select our data chunk size as 8 elements. So we need thread block with 8 threads as well. In the first iteration we set offset to 1. And in this iteration only T0, T2, T4, T6 threads are going to perform the summation. T0 thread is going to sum up elements in 0th index in the input array with element 1 offset away from 0th index which is the element in the first index and store the results back to 0th index in original array. T2 is going to sum up next two elements and store the results back to index 2 in the array. T4 is going to sum up next two elements and store the results back to index 4 in the same array. And T6 is going to sum up final two elements in the data chunk and store the results back to index 6 in the input array. So, after the first iteration, our data chunk would look like this. Notice only half of the threads in the thread block did any effective work in the first iteration and also, we store the summation again to the memory that we are load from. This kind of accumulation referred to as in-place reduction where we store the memory to the same location we load the memory from. Input to the second iteration would be the result calculated from the first iteration. So first element in the array now contains the summation of first and second element in the original array. 3rd element in the array for the second iteration have the summation of 3rd and 4th element in the original array and so on. In the second iteration we are going to sum elements which are in twice the distance than in first iteration. So the offset value will be 2. And only 2 threads are going to perform any effective work in this iteration. T0 and T4 threads. T0 will sum up 1st and 3rd elements in the array and store it back to 1st element again. Notice after this calculation first element value is the summation of the first four element in the original array. In the original array we have 3, 5, 1, 2 as first four elements. which is sum up to 11, and that's what we got as the first element after second iteration. And T4 will sum up 5th and 7th element in the array and store it back to the 5th element. And this value correspond to the summation of second 4 elements in the original array. In the next iteration our offset value will be twice as previous iteration. So we have 4 as our offset size and only one thread, T0 is going to perform the summation. Now input to this iterations is the output from previous iteration, so when T0 sum up the 1st and 5th element in the input array, output array is the summation of all element in the original data chunk. So after 3rd iteration we have the summation of all 8 elements in the data chunk as the first element in the data chunk. Then we will store this value in separate array which contains the partial sum like this for all the data chunks we made. Code segment for summing up data chunk in neighbored pairs approach is shown in this slide. As I mentioned earlier, we are going to start the iteration with offset as one and then we are going to multiply the offset value in each iteration. And with in each iteration we have to limit which thread is going to perform the summation. For example, in the first iteration, T0, T2, T4, T6 threads did the summation. And in the second iterations only T0 and T4 threads perform the summation. So to accommodate this kind of limitation, we have to check the thread id before performing the summation. This condition check guarantee that only necessary threads in each iteration are going to execute the summation. For example, in the 1st iteration, our offset value will be one, so the threads with even number thread ids will only be able to execute the summation and in the second iteration offset value is 2, so the threads with thread id which is a multiplication of 4 will be able to execute the summation and so on. If you notice in this implementation we have to make sure that all the threads in the block should finish execution of one iterations before any of threads move on to the next iteration. For example, if we have 128 size thread block, then we will have 4 warps and all 4 warps have to finish 1st iteration before moving on to the next iteration. But warps can execute in any order. So when we have thread block with multiple warps we don't have any way of guarantee this restriction. This is where syncthreads() function comes to the action. After each iteration, we are going to use synctrhead() function. So every thread in the thread block have to wait until all the threads in that particular thread block finished this iteration. Ok let's see the implementation now. As a practice, we are going to check the validity of calculated results from the GPU implementation by comparing it with value got from this CPU implementation. So here we need one function to perform the reduction in CPU and one function to compare the results of CPU and GPU implementations. And as I told you, neighbored pair is only one way of implementing parallel reduction, and in upcoming videos we are going to discuss more ways of implementing parallel reduction. So if I put this function in a common place then in the upcoming implementations we can access these functions again and again. So here I have a common header which include such common definitions. So let's add these functions to the common header. First we have to declare reduction\_cpu function in common header. Then we can define the function in common.cpp file. Here, all you have to do, is iterate through array while adding each element to the global variable in this way. I Compare result function, will perform comparing of given two integers for equality. So let's declare this function in common header file as well. then we can define it in common.cpp file. We can first print out the both values is being compared, then if the values are equal we can print out the values equal to the console and if the values are different we can print out value are different to the console. Now to our reduction file. Let's name our kernel as reduction neighbored pairs and it will take 3 arguments. Input array and the array to store partial sums and the size of the input array. We will get back to kernel implementation shortly, but now let's ,move on to our main function. Now in most of our implementations our main function will look similar. So here, I am going to completely explain to you what we are going to do in main function, but in upcoming videos we will not focus on main function, so we can simply focus on our kernel implementations. In the main function, first, we are going to define our size variable and byte size variable which is the actual number of bytes taken by our input array, which in this case 128 mega bytes of data, and our block size will be 128 threads. And then we need to declare a pointer to hold input array and host side pointer to partial sum array and allocate memory for those pointers. Note that here we are going to initialize our input array randomly with value 0 to 10. For that I have use initialized function defined in the common header. And then we can calculate CPU results and store it to the variable called CPU results. Then we need our kernel launch parameters block and grid variables. So let me calculate those values as well. And then usually I'm printing out the grid and block size parameters, so in the output we can clearly identify our grid configuration. If you notice, I have not yet allocate memory for partial sum array. Size of the partial sum array is equal to number of blocks in our grid and only now we know that. So here let me take the number of bytes needed to hold partial sum array and then we can allocate memory for that. Now we need device pointers. So let me declare those pointers here. And then we can allocate memory for those pointer using cudaMalloc() function with particular pointer and size of memory we need. Next we have to set initial value to partial sum array to 0. . So here we can use cudaMemset() function for that. And then we need to transfer the input array to device from host. So here I am going to use cudaMemCpy() function with necessary pointers and our memory copy direction is host to device in this statement. Then we can launch the kernel with given arguments and after that we have to wait until our kernel to finish, so here, we can use the cudaDeviceSynchronize() function we discussed in the presentation. And then we can transfer the partial sum array back to host. Now to calculate the final results from the partial sum array, we have to iterate through partial sum array and add it to global variable. So here , I am going to iterate through all the elements and perform that summation. And then we can check the validity of results by passing both CPU calculated results and GPU calculated results for our previously defined compared results function. And finally we can clean up the memory we alocated in host and device by calling cudaFree and free functions. Ok that's it for the main function, And in the upcoming implementation our main function will look most likely these, so I'm not going to explain there. Now let's look at our kernel implementation. In the kernel we need two variables tid and gid to hold thread id in block and global id of a thread in a grid so let me defined those and assign those values. And then we need to perform boundary check to avoid invalid memory access from the threads. And then we have for loop I shown in the pseudo code. Here we have set initial offset value to 1 and then we are going to multiply it by 2 in each iteration until the offset is less than the block size in X dimension. And then we have a if condition to filter the threads which can perform summation for given iteration. And after we perform summation, we need synchthread() function call. So all the threads in a block have to finish one iteration before any of threads in that particular thread block execute the next iteration. And after all the iteration ends in every data block first elements have the summation of that block. So we can assign that value to our partial sum array. Now the index of this partial sum array is equal to the block id for particular block. So the zeroth thread in each block will each block will store the sum of that block in to the partial sum array in this way. That's it for our kernel implementation. Now let's run this program. Ok, in the output it printed out that results are same. Which means both CPU and GPU calculation yeild the same results. which means our implementation of reduction algorithm with CUDA is correct. Ok, now you are aware of how to use syncthreads() function to synchronize the threads with in a block. But you have to be careful when you are using syncthread function. Specially if your code contains condition checks. We perform syncthreads() function call to make sure that all the threads in a block reach one point before any of the thread in particular block moves forward. But if we are using syncthreads with in a condition check then that condition check might not allow some of the threads in that particular thread block to reach the syncthread() statement based on the condition.

This will result in paradox and the outcome will be undefined value. So be mindful when you are using syncthreads() function specially in the presence of condition checks.