CUDA Dynamic parallelism:

In this video we are going to discuss about a concept called Dynamic parallelism in CUDA programming. Dynamic parallelism was introduced to CUDA paradigm with CUDA toolkit 5.0 and you are going to need device with at least 3.5 compute capability to use the dynamic parallelism. So what is dynamic parallelism. Well so far in this course, all kernels have been invoked from the host thread. The GPU workload is completely under the control of CPU. But CUDA dynamic parallelism allows new GPU kernels to be created and synchronize directly on the GPU. So we can launched the kernels from another kernel in same way we used to launch the kernels in host applications. So what are the benefits of having this facility. Well with dynamic parallelism, You can postpone the decision of exactly how many blocks and grids to create on a GPU until the runtime. And use of dynamic parallelism can make your recursive algorithm more transparent and easier to understand. Also the ability to create work directly from the GPU can also reduce the need to transfer execution control and data between host and device as launch configurations decision can be made at runtime by threads executing on the device. In dynamic parallelism kernel executions are classified in to two types. Parents and Childs. Parent grid is configured and launched by the host thread. And the child grid is configured and launched by the parent grid. Consider the diagram shown here. As highlighted in this diagram, parent grid is the one start the execution and also it will be the one end the execution of the program. If we do not explicitly provide any synchronization then implicit synchronisation will happen to adhere , this restriction. In case of multiple level of recursive calls, kernels in the middle of recursive hierarchy can also refer to as parent with respect to kernels which are launched from those kernels. Grid launches in device threads are visible across thread block. This means that a thread may synchronize on a child grid launched by that thread or other threads in the same thread block. Execution of a thread block is not considered complete until all the child grids created by all the threads in the block have completed. If all the threads in the block exist before all the child grid have completed implicit synchronization on those child grids is triggered. When a parent launches a child grid the child is not guaranteed to be begin execution until the parent thread block explicitly synchronized on the child. Parent and child grid shared the sane global and constant memory storage. But have distinct local and shared memory. Parent and child grid have concurrent access to global memory. There are two points in the execution of child grid when its view of memory is fully consistent with the parent thread. At the start of the child grid and when the child grid completes. All the global memory operations in the parent thread prior to child grid invocations are guaranteed to be visible to the child grid. All memory operation of child grid are guaranteed to be visible to the parent after the parent has synchronize on the child grid's completion. It is invalid to pass pointer to local memory as an argument when launching a child grid. Ok, With these concepts in mind let's see couple of example of use of dynamic parallelism. In our first example, we are going to print out the hello world to console with recursive kernel launches. Here are the specifications for our program. Parent kernel will be launched from host with 1 thread block having 16 threads. First thread in this grid has to launch child grid which has half of the elements in the parent grid. Child grids minimum element count is one. Each thread in the grid should print out threadIdx.x value and the depth of the grid. Consider a parent grid's depth as 0 and increase the depth by one in each recursive call. Ok, let's try to implement a cuda program to adhere these specifications. Let's start with the signature of our kernel. Well in the specification it mention us to reduce the grid size by half in each recursive launch so we need a way to transfer grid size to recursive call. So we need a parameter for that. Let's name it as size. Then we also need to print out the depth of the recursion, so let's have another parameter called depth as well. Let's leave the kernel in this state for now and move on to main function. In the main function we need to launch a grid with 16 threads and 1 thread block. So our kernel launch parameters will be like this. And for parent grid, size is 16 and depth is 0. So we can pass parameters for kernel launch like this. Now back to our kernel. In each recursive call we have to print thread id and depth of the recursion call. So let me printed that as the first thing in the kernel. then if the grid size is 1, then we have to stop the recursion. So let me add that stop condition here. if not, we have to launch another grid in first thread in the this thread block. So if the thread id is 0, we have to launch grid which has half of the size as current grid. Also we have to pass the launch grid size and depth as parameters for this kernel launch as well. That's it for the implementation. Now let's compile this program with nvcc and run this program. Now dynamic parallelism is allowed in the device with compute capability 3.5 or higher and dynamic parallelism also needs relocatable codes. So we have to specify these two options to nvcc compiler here. First use -arch option or architecture option to specify the compute capability in this case sm\_35 or higher value and set -rdc option to true. After you compile with these options you can run the program. Now in the output you can see that in the 0th depth if has printed out 16 statements. And when depth increases statements are reducing by half. So at depth 1, there are 8 statements. At depth 2 there are 4 threads, so the print out statement has been print out 4 times and so on. Now I am going to show you the completion order of these recursive kernel execution. For that I am going to use nvvp or nvidia visual profiler. We will discuss about this tool in detail in upcoming sections, but for now let's use this tool to visualize our kernel execution. So execute the command nvvp with application name in this way. This command will launch nvvp profiler and after couple of seconds you will be greeted with window to select workspace. Just click Ok here. Then again, another dialog call executable property dialog will pop up and there click next at the bottom and in the next step click finish. Ok, now you can see the execution timeline for your program. Now due to the smaller execution time of you kernels, you will not be able to see the kernel execution clearly first. So hold control key and scroll down to zoom up the timeline. Then you will be end up in view like this. On the left side you can see the kernel execution hierarchy. There you can see that first kernel to execute or parent kernel In this case consume 100 percent of runtime. This is the proof that all the child has to complete before parent kernel completes. You can observe this fact using timeline as well. As you can see here, even though parent kernel executes their body quite early, still those has to wait until child kernels to finish. The child grids are properly nested and each parent grids wait until it's child grids are completed. Here white space are used to indicate a kernel spending time waiting for child to complete. Observe you nvvp timeline and you will realize these facts by your self. That's it for this video. But before moving on the next video you have to perform simple modifications to this program we have here to achieve following kernel execution arrangements. The firs one is this. You have yo launch a grid with 2 thread blocks each having 8 threads from the host. Then from each thread block you have to launch child grid with half of the block size as we did in this example we discussed in this video. Minimum child grid size is 1, So the hierarchy of grids would resemble the diagram shown in this slide. This second one is this. Here also we launch grid with 2 thread blocks with 8 threads from the host, and in the first thread block each grid has to launch two child blocks with half of the size of parent thread block until the thread block size become 1. Each thread in these grids has to printout threadIdx.x , blockIdx.x and depth values. So submit the output of the program execution for these two arrangement in the next assignment.