Profiling with nvprof:

Profiling is a very important aspect of CUDA programming which gives us insight about our program execution based on different grid configurations and different resource usages. In this video, we are going to discuss about standard profiling tools comes with CUDA tool kit called nvprof. Now in upcoming CUDA tool section we will give you in-depth insights on how to use nvprof and couple of more tools as well. The focus of this video is to give a head start on nvprof, so that you can use the tool throughout the course. Different usages of nvprof will be discussed throughout this course. But in this video we will only discuss what we need to understand profile driven optimization concepts. In profile driven optimization . We change our program based on the profiling output . Concretely in this video you are going to implement array summation in GPU. Here, we will provide different implementation based on different grid configurations and we will observe the performance variations based on these configuration via nvprof tool. Before explaining the tool, Let me first show you the program we are going to profile here. The program we are going to use here is the sum\_array program which we implemented in the previous section with some modifications. I have modified that program to take command line arguments. Now this is quite a common approach, when we are using programm profiling, as we need to change the grid configurations and resource usages like shared memory dynamically. If we haven't used this type of mechanism then to change grid configurations or resource usages we have to manually change code and compile it again and again before profiling. Ok, in this kernel, as the argument we will provide whether we need our grid to be 1D or 2D grid. . As the second argument we provide the input size of the array. These arrays will be randomly initialized. In case of two dimensional grid, we can provide the number of thread in X dimension as 3rd argument. And as 4th argument we will be providing block size in X dimension and 5th argument will be the block size in Y dimension in case of two dimensional blocks. Now these size value will be given as power of 2. For example, if we need 1 megabyte of data which is 2 to the power 20 bytes, so we can give 20 as the second argument. The implementation is pretty simple. The difference here is the addition of condition checks to direct the flow of the main program to decide which type of kernel its going to execute. So I encourage you to go through the code for this video after watching the video. And also I have use result validation with CPU as well. So we can validate our results. But I have not include any timing related codes here, since we are using these codes to profiling purposes. Note when profiling more additional information on execution will be recorded by the profiler. So it is not recommend to time your kernel while profiling since profiling will add pretty big overhead to the instruction execution. Especially nvprof should not use to compare the execution time with CPU implementations. But if you have couple of kernels, and if you need to compare those two kernels, then you can use nvprof. With these things in mind let's move on to our main focus of this video which is the profiling with nvprof. Nvprof have 3 modes of operations. Summary mode, gpu and api trace mode, event metrics summary mode, and event metrics trace mode. Out of these, right now we will focus on summary mode and event metrics summary mode only. But throughout the span of this course depending on the needs, we will use other modes of nvprof as well. Our first step is to compile the CUDA program we are going to profile. So let me do that now. Now since we are going to measure timing as well, keep in mind not to include any flags which will add overhead to the executable code. For example don't compile the code with -g or -G flags which will disable the compiler optimizations. After that you can use syntax shown in the presentation to profile the application. So for the nvprof tool, first we can give options like metrics, events so on. Then we have to specify the application and the application argument. Let me run nvprof with our program without any options. The outcome is in summary mode. Summary mode is the default operation mode for nvprof. In this mode nvprof outputs a single result line for each kernel functions and each type of CUDA memory copy perform by the application. Let me explain this output. Now we have not specified any options to nvprof and that's why we got the default output or output in summary mode. Also we have not specified and arguments to our program as well. So here our application use default size and our kernel launched with 1D grid. Also, you can see that results of GPU and CPU are same, which verify the validity of execution we just did. In the summary mode, you can see the GPU activity section first. Kernel executions, memory copy functions will be categorized under this section. Now we have transferred two arrays to device from the host. And you can see that in output, there are two CUDA memory copy calls happen in host to device direaction. That operation took 54 percentage of the total time of execution. Now we have 2 CUDA memory copy function in host to device direction. So here nvprof records the minimum value of those calls and maximum value and the average values for those calls. But we did only single memory copy in device to host direction. So all the values: min, max and average values are same. And we have our kernel execution, which took only 2% amount of total execution time. Now you can see the significance of memory transferring overhead here. And then you can see the API calls and there profiling details as well. Some of these functions happen underneath to accommodate what we done in the code. To have the output in event metric summary mode we have to specify these options to nvprof tool. For example let's say I need to look at some metrics for my application. Metrics are different measurements of your CUDA applications like warp divergence, efficiency of SM, global memory load efficiency and etc. This slide highlights few of the important metrics we are going to use throughout this course. We will discuss event parts in upcoming section. Ok let's look at global memory load efficiency SM efficiency, and achieved\_occupancy for different configurations of our application. First let's run all application with default configuration which is one dimensional grid with block having 128 threads. In the command prompt, you have to specify the metrics you want to measure using double dash metrics option and metrics should be comma separated. Ok, let me run this command now. Now as you can see from the output, it has list metrics names with metric description and values for corresponding metrics. In this case we have 100% efficiency in global memory loads and SM efficiency is almost 99% and we have 0.89 or 89% achieved occupancy for the default configuration. Now here is what we are going to do, we are going to increase our array size to 32 megabytes which is 2 to the power 25 number of bytes and then we are going to profile the sum array application using following grid configurations while reporting the profiling details. In the first configuration we are going to arrange all the threads in to 1D grid. Block size is 128 threads in X dimension. So the arguments to our application will be 0,25,0,7. First 0 to specify whether we need 1D grid or not, 32 megabytes means 2 to the power 25, so we have 25 as second argument. Since this is one dimensional grid we will not worry about size in X dimension as it is same as the input size, And then block size in X dimension which is 128 equals to the 2 to the power 7 So, 7 is the next argument. In second configuration, we are going to have 2D grid with dimension shown in this diagram. Here, also we have 32 megabytes of data and we are going to arrange our grid to have 2 to the power 20 threads in X dimension and 32 or 2 to the power 5 threads in Y dimension. And our block size will be 128 threads in X dimension and 4 threads in Y dimension. Now keep in mind our max block size is 1024. So you how to make sure that your block size is not exceed the 1024 thread per block restriction. So, in this case, our arguments will be 1, 25, 20, 7 and 2. Next configuration is almost similar to the second one. The only difference is that I have change the book size in X dimension value to 256.. So in 3rd configuration we have 256 \* 4 threads which is 1024 threads per block. Now this is the maximum value that thread block can have. If you increase this value further then you're kernel launch will be failed. Now let me run our profiler with each of these arguments set. Let's run one dimensional grid first. Now as you can see, nvprof has given us the information about metrics. Now let's run two dimensional configuration. And finally let's run the 2D configuration with 256 threads in threads block's X dimension. Now if you compare these outcomes you can see that occupancy values has been different for each of these launches. Ok, My purpose of this video was to show you how to use a nvprof tool for profiling purpose so you can have insight about your program and we are going to use this tool extensively in upcoming lectures for profiling purposes. And if you want to learn more about nvprof you can type nvprof help command. So if you run that command, you can see from the output, it has all the details about commands options metrics and events.