CS 629/729 GPU Programming
Lab 4

Spring 2024

Learning Outcomes

- How to query CUDA device properties
- Understanding how to observe the difference between theoretical and measure memory bandwidth
- Understanding how to design a basic ray tracing application
- Understanding how to use the Cooperative Groups feature

Task 1

In task 1 we are going to extend our vector addition kernel. The code is provided in task01.cu. Complete the following changes

1.1 Modify the example to use statically defined global variables (i.e. where the size is declared at compile time and you do not need to use `cudaMalloc`). *Note: A device symbol (statically defined CUDA memory) is not the same as a device address in the host code. Passing a symbol as an argument to the kernel launch will cause invalid memory accesses in the kernel.*

1.2 Modify the code to record timing data of the kernel execution. Print this data to the console.

1.3 We would like to query the device properties so that we can calculate the theoretical memory bandwidth of the device. The formal for theoretical bandwidth is given by;

$$theoreticalBW = memoryClockRate * memoryBusWidth * 2$$

Using `cudaDeviceProp` query the two values from the first `cudaDevice` available and multiply these by two (as DDR memory is double pumped, hence the name) to calculate the theoretical bandwidth. Print the theoretical bandwidth to the console in GB/s (Giga Bytes per second). Note that the above will calculate the result in kilobits/second (as $memoryClockRate$ is measured in kilohertz and $memoryBusWidth$ is measured in bits). You will need to convert to the memory clock rate to Gb/s (Gigabits per second) and then convert this to GB/s.

1.4 Theoretical bandwidth is the maximum bandwidth we could achieve in ideal conditions. We will learn more about improving bandwidth in later lectures. For now, we would like to calculate the measure bandwidth of the `vectorAdd` kernel. Measure bandwidth is given by:

$$measuredBW = \frac{R_{Bytes} + W_{Bytes}}{t}$$

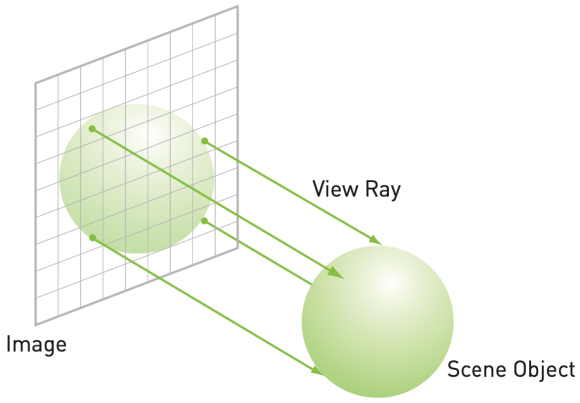
Where R_{Bytes} is the number of bytes read and W_{Bytes} is the number of bytes written by the kernel. You can calculate these values by considering how many bytes the kernel reads and writes and multiplying it by the number of threads that are launched. The value t is given by your timing data in ms you will need to convert this to seconds to give the bandwidth in GB/s. Print the value to the console so that you can compare it with the theoretical bandwidth.

Task 2

For this task we are going to implement a simple ray tracer application. Ray tracing is one effective way of producing a 2D image of a scene consisting of 3D objects. It works by tracing a path from an imaginary eye through each pixel in a 2D image plane, and calculating the color of the object visible through it. Typically, each ray must be tested for intersection with some subset of all the objects in the scene. Once the nearest object has been identified, the algorithm will estimate the incoming light at the point of intersection, examine the material properties of the object, and combine this information to calculate the final color of the pixel.

The real ray tracer is a lot more complicated, so we are going to implement a simplified version. In our simplified version, we are going to create a 2048*2048 2D image. The 3D scene will contain only randomly generated spheres each has random center position (x, y, z), color (r, g, b), and radius. See details in the `Sphere` struct.

Our imaginary eye is restricted to the z-axis facing the origin, so for each ray shooting from pixel (x, y), the direction of each ray is $\vec{r} = (x, y, 0)$. z is 0 because we assume all rays are in parallel with the z-axis.



Your program should check all possible intersections between every ray and every sphere.

For example, if you have 16 spheres in the 3D scene, you should check the potential intersection 2048*2048*16 time.

If a ray $\vec{r} = (x, y, 0)$ intersects with a sphere centered at $(x_{shpere}, y_{shpere}, z_{shpere})$, then:

$$\sqrt{(x_{shpere} - r.x)^2 + (y_{shpere} - r.y)^2} < radius$$

The z coordinate of intersection position is:

$$z_{shpere} - \sqrt{radius^2 - \left(x_{shpere} - r.x\right)^2 - (y_{shpere} - r.y)^2}$$

If a ray interests with multiple sphere, you should use the z coordinate of the intersections to determine which sphere is closest (smallest), since only that sphere is visible.

The color of the pixel should be determined by the color of the visible sphere and where the ray hit the sphere. You should first calculate the color ratio which controls the saturation of the color based on where the ray hit the sphere:

$$color_{ratio} = \frac{\sqrt{radius^2 - \left(x_{shpere} - r.x\right)^2 - (y_{shpere} - r.y)^2}}{radius}$$

The actual color of the pixel should be calculated as follows based on color of sphere that is visible.

$$(color_{ratio} * r, color_{ratio} * g, color_{ratio} * b)$$

The provided code (task02.cu) contains the structure of the ray tracer application with testing functions. To complete the task, you need to implement the ray_trace kernel. The ray_trace kernel is designed to parallelize the workload of each ray that started from each pixel position in the 2D image. You should implement the logic of determine if the current ray intersect with any of the spheres in the 3D scene. If so, which spheres it interests with and what would be the color for the pixel. Save the color in the r, g, b, variable that will be saved to the final image data array. When running the code, the testing code will try to generate different number of spheres (from 16 to 2048) and call the ray_trace kernel to generate images. Each image will be saved in ppm file that can be viewed using Mac Preview or Photoshop. Report the time takes for each different sphere count in the following table.

Sphere Count	Time
16	
32	
64	
128	
256	
1024	
2048	

Task 3 (CS 729 only)

In the lecture, we introduced the reduce sum algorithm using thread block level synchronization (method 2) and grid level synchronization of Cooperative Groups (thread 3). Create a new source code file named task03.cu with the following

1. Implement the two algorithms for reduce sum.
    a. For method 2, you should have a kernel that works one level of the reduction tree, so you need to call your kernel multiple times to finish the reduction tree (until reduced to only one value).
    b. For method 3, you should have only one kernel and call the kernel once for handling all levels of a reduction tree. Because cooperative groups can only allow launching a certain number of threads in a grid (e.g., N), if the total number of input values exceed N*2, you need to partition the input the data into multiple reduction trees and use a loop inside the kernel to sequentially process each tree and have one thread block reduce the local sum of each tree in the very end.
2. Design a testing code for the two kernels that include:
    a. Randomly generate input arrays of float type
    b. Testing input arrays of size 1M, 2M, 4M, 8M, and 16M (M = million)
    c. A CPU validation function
    d. Measure and report the time cost for each method with different input sizes.