UNIVERSITY OF SOUTHERN CALIFORNIA

Ambient Occlusion Ray Tracing on the GPU

Stephen Hopkins and Guanghui Chen
Computer Science

Abstract: Our project implements a parallel ray tracer featuring ambient occlusion on the GPU. Taking advantage of the GPU's massively parallel processing structure, we can quickly render an otherwise slow CPU ambient occlusion render pass. In our results, we compare the performance of constant memory versus global memory for storing the scene.

Method

General Ray Tracing

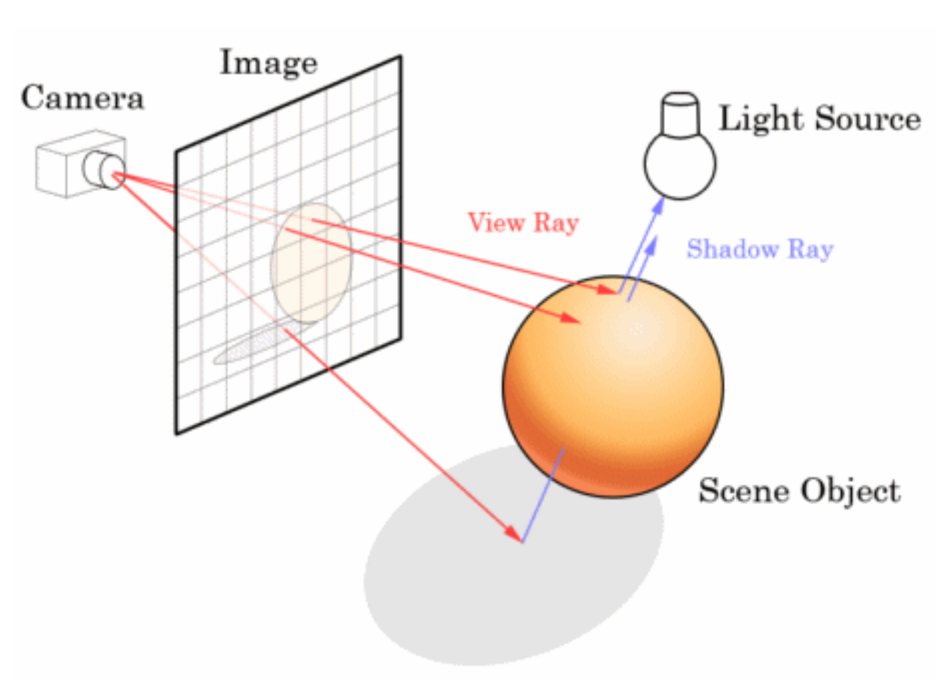

Fig. 1 - The rays start from the camera and pass through the "Image" to hit the scene objects and determine the appearance of pixels on the screen.

By simulating the propagation of light rays, ray tracing generates virtual rays and uses them to intersect with geometries within the scene.

For each pixel on the screen, or "Image" in *Fig. 1*, the ray tracing engine generates single or multiple rays. When a ray hits geometry in the scene, the color calculated at that hit point will be printed in the "Image".

Additionally, with ray hitpoints and surface normals, secondary computations can be done such as shadows, reflections, and ambient occlusion. These secondary calculated colors are then composited together with the initial diffuse model into a final color value. Our project focuses on ambient occlusion which is described below.

Ambient Occlusion

Ambient occlusion (AO) is a method in computer graphics to approximate the way light radiates in real life. It can determine a surface bright or dark based on how easy it is for the surface to 'view' the outside world and does not need any lighting information.

In our implementation, we calculate ambient occlusion by sampling random rays from the hit point and counting how many rays are blocked. The ray direction for each sample is within the halfspace defined by the surface normal, **n** in Fig. 2.

This is an expensive secondary ray tracing operation, depending on how many ray samples are used, so we accelerate it by doing it on the GPU with CUDA.

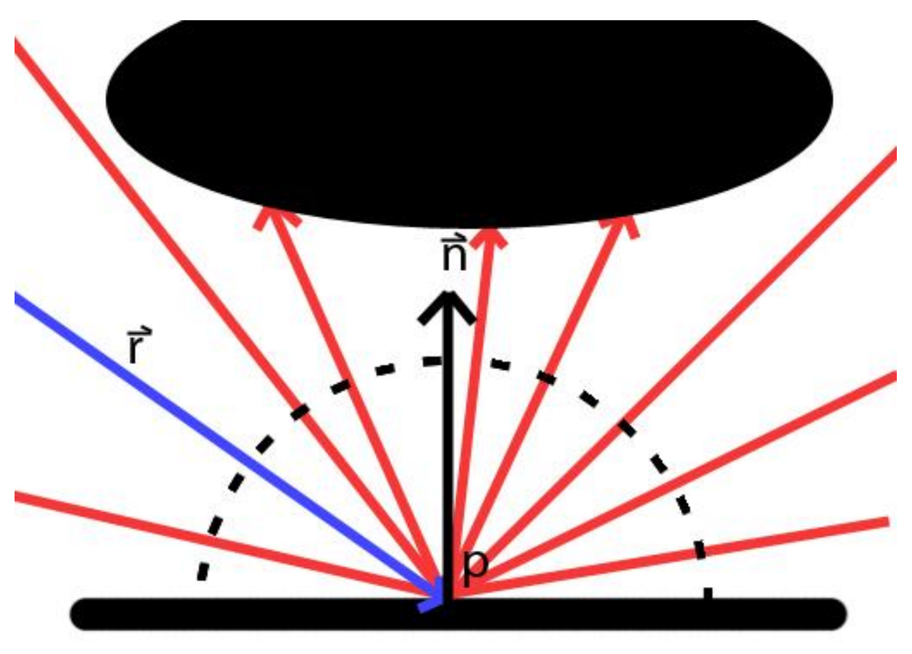

Fig. 2 - Image ray **r** intersects at point **p** on surface with normal **n.** The red vectors are the random sample rays generated by the AO algorithm

Parallel Algorithm for CUDA

width - the width of the output image
height - the height of the output image
maxDistance - the maximum distance for a ray that counts as an intersection
numSamples - the number of secondary rays to send out for calculating AO

```
pixelX = blockIdx.x * blockDim.x + threadIdx.x
pixelY = blockIdx.y * blockDim.y + threadIdx.y
for(i = pixelX; i < width; i += gridDim.x * blockDim.x)
   for(j = pixelY; j < height; j += gridDim.y * blockDim.y)
      calculate ray from (pixelX, pixelY) and cameraPos
      intersect ray with scene to obtain intersection point, p, and normal, n
      numHits = 0
      for k = 0 to numSamples
         send ray in random direction within the hemisphere defined by p and n
         if(ray intersects with scene within maxDistance)
            ++numHits
            output[pixeY * width + pixelX] = numHits / numSamples
```

On the left is the pseudo code for the kernel function on the GPU. The idea is to distribute ray tracing to each thread available to the GPU and cover every pixel on the image. See *Fig. 3*.

The distribution takes into account both x and y positions on the screen and it can aggregate the pixels with similar geometry hits to the same block on GPU, and thus achieves better locality in memory access.

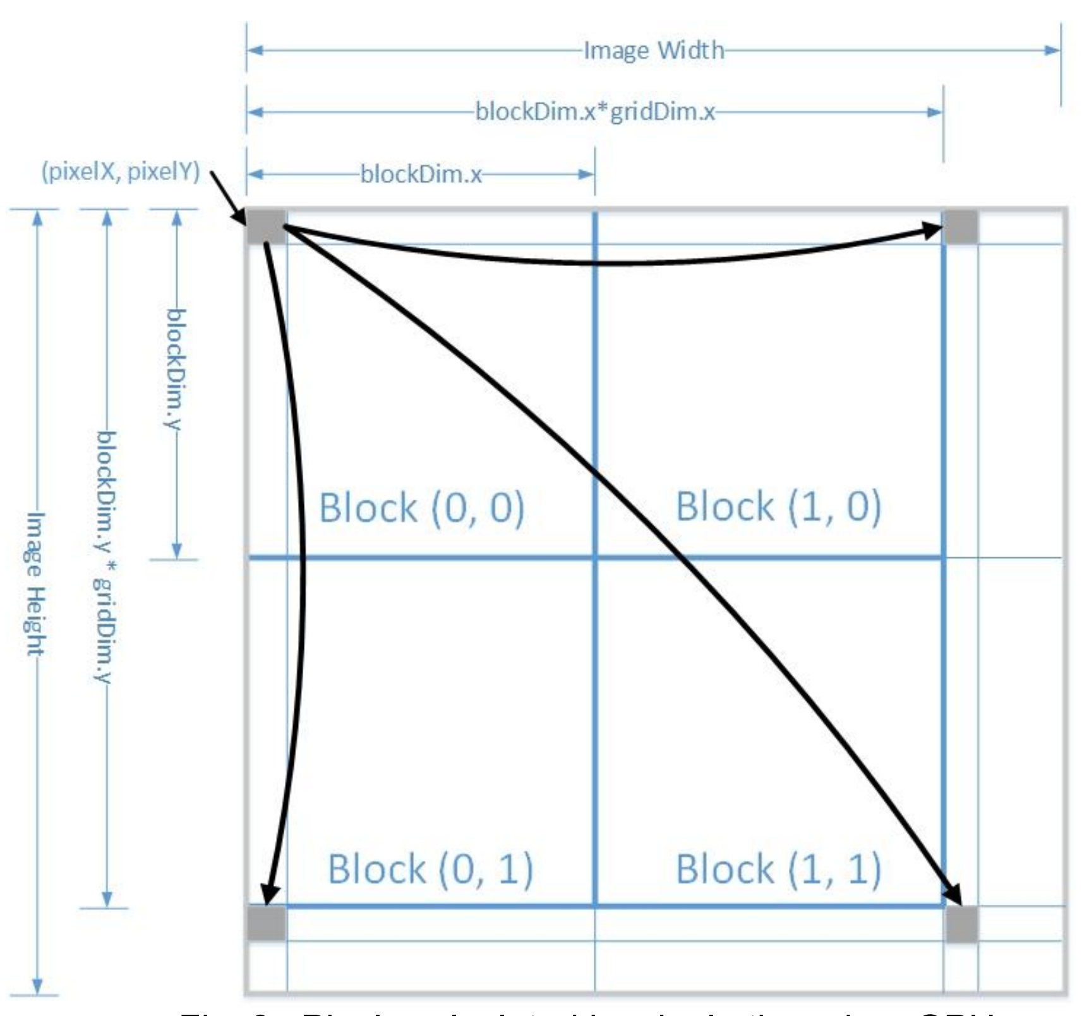

Fig. 3 - Pixels calculated by single thread on GPU

GPU Memory Performance Test

The threads on the GPU are arranged in blocks and the blocks are arranged in a grid. These different layers of organization lead to a hierarchical memory structure. Global memory and constant memory can be accessed by any thread on the GPU. On the other hand, local memory is accessible block wide, or per thread.

We try two types of memory usage strategies. One is to put as much data in constant memory as possible. The other way, we only utilize the global memory. With these two implementations, we are able to compare the performance of each type of memory on the GPU through the performance of the ray tracer.

Results

References
Real-time ray tracing with CUDA, M. Shih, et al., in *Proc. of Int'l Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP '09)* (2009)
Raytracing diagram in Fig1. from http://www.codinghorror.com/blog/2008/03/real-time-raytracing.html