CSED490V Programming Assignment 2

Georg Boehm - 49003999

November 26, 2022

Contents

1 Problem Definition

The task in this assignment was to write a parallel circle renderer with cuda. The renderer accepts an array of circles (3D position, velocity, radius, color.

Figure 1 illustrates the basic algorithm for computing circle-pixel coverage using point-in-circle tests. These tests will become relevant in the later implementation as well. All pixels within the circle's bounding box are tested for coverage. For each pixel in the bounding box, the pixel in considered to be covered by the circle if its center point (shown with a dot) is contained within the circle. Pixel centers that are inside the circle are colored with the corresponding color of the circle, while those that are outside the circle will not.

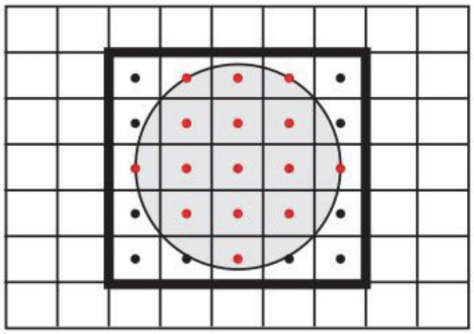

Figure 1: Contribution of circle to image.

The renderer renders semi-transparent circles, meaning the color of any one pixel is not the color of a single circle, but the result of blending the contributions of all the semitransparent circles overlapping this specific pixel.

In conclusion, our parallel implementation has to perform updates to an image pixel in circle input order. That is, if circle 1 and circle 2 both contribute to pixel P, any image updates to P due to circle 1 must be applied to the image before updates to P due to circle 2.

2 Intuitive solution and limitations

Since we have to perform updates to any pixel in the image in the correct order, our parallel solution does not become trivial. We have to make sure our threads work together in a manner that update operations will still be calculated in parallel but the updates themselves will be executed in the correct order. We also have to make sure that no two threads update the color of the same pixel at the same time (atomicity).

One intuitive solution would be to assign each thread to one pixel in the image, calculate which circles overlap with this pixel and color in the case of overlapping. That will achieve a correct coloring of each pixel. The circles are indexed 1 through n (with n being the total number of circles in the image).

This solution is rather slow, since we process every single circle per thread, only utilizing parallelism to a very small degree. We would also have to access global memory for each of our threads to get the list of all circles per thread. We have to come up with a different solution.

- Describe your intuition briefly
- Different axes for parallelism (across pixels, across circles)
- Data reuse where?

3 Proposed solution

- How does your solution work (broadly speaking)

It is important to notice that we can achieve atomicity for the image update operations by letting each thread process one pixel individually. We therefore just have to ensure the correct ordering of pixel update operations per pixel. This we can also solve by utilizing thread blocks. Therefore, each thread is assigned to one pixel and multiple circles throughout different parts of the coloring process. In the following we will discover the broad scheme of the proposed solution in this report. A more detailed solution can be found in section 4.

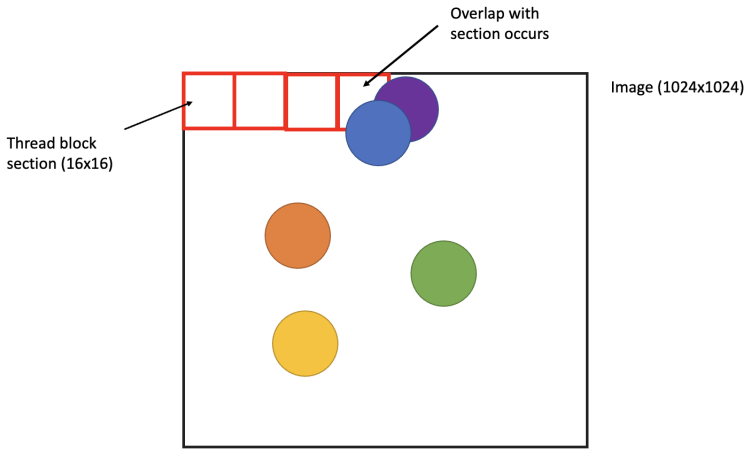

Figure 2: Broad idea of the proposed solution: Tile image into 16x16 sections, assigning each 16x16 thread block to one of these sections. Let the threads in each section work together to identify the circle indices overlapping with that specific section. Then assign each thread to one pixel in the section and check if this pixel overlaps with the circle in the section. Color if that is the case.

We basically tile our input image (usually we are speaking of size 1024x1024) in smaller subsections and assign each section to a thread block. We then let the threads in this thread block work together to find which circles lie in this section. After identifying the correct indices of these circles, we can now simply let every pixel in our section check if it overlaps with these few circles or not. This is still a bit costly, since a pixel might be checking if it overlaps with X amount of circles (X being the length of the list of circles overlapping with the section the pixel lies in) while not overlapping with any of these circles at all. However, there is not really an efficient way around this. Minimizing the amount

of circles to check per pixel provides a much higher performance itself. The basic idea of this solution is depicted in figure 2.

This solution is not only faster than the intuitive solution, but also fulfils the atomicity and correct ordering for the coloring operations. As we will see in the detailed solution, we do not have to make adaptations to the operation for shading the pixel with our proposed solution.

4 Detailed solution

4.1 Thread blocks

Each thread block consists of 16x16 threads. Working with 2-dimensional thread blocks make it easier to understand and implement the tiling procedures for our 2-dimensional image. Each thread block covers 16x16 pixels of the image.

4.2 Identify circles in section

We utilize the provided circleBoxTests by the course instructor to identify which circles overlap with a given section. We therefore have to define the box (section) for each threadblock needed for the circleInBox() test.

```
// p == cuConstRendererParams
float boxL = static_cast<float>(blockIdx.x) / static_cast<float>(gridDim.x);
float boxR = boxL + static_cast<float>(blockDim.x) / static_cast<float>(p.imageWidth);
float boxB = static_cast<float>(blockIdx.y) / static_cast<float>(gridDim.y);
float boxT = boxB + static_cast<float>(blockDim.y) / static_cast<float>(p.imageHeight);
```

We now need to loop over every circle and check if it overlaps with this section. The tests will return a 1 if a given circle overlaps with the section and 0 if it doesn't. Therefore, we check 256 circles per iteration for overlap with our section.

```
// scanInput == binary array of length 256
for (int i = 0; i < cuConstRendererParams.numCircles; i += BLOCK_SIZE) {
    int circleIndex = threadIndex + i;
     // Find circles in box
    circlesInBox(circleIndex, threadIndex, scanInput, boxL, boxR, boxT,
       boxB);
    __syncthreads();
    // TODO: Perform inclusive scan on scanInput
    // TODO: Get circle ids
    // TODO: Perform coloring
}

// circleInBox function
__inline__ __device__ void
circlesInBox(size_t circleIndex, size_t threadIndex, uint* inputArray,
   float boxL, float boxR, float boxT, float boxB) {
    if (circleIndex < cuConstRendererParams.numCircles) {
        // int index3 = 3 * circleIndex;
        float3 p = *(float3*)(&cuConstRendererParams.position[circleIndex
            * 3]);
        float rad = cuConstRendererParams.radius[circleIndex];
        // check if circle is in box
        inputArray[threadIndex] = circleInBox(p.x, p.y, rad, boxL, boxR,
           boxT, boxB);
    }
    else { // can impossibly be in box
        inputArray[threadIndex] = 0;
    }
```

This will return us a binary array of length 256 (refer to **scanInput** in figure 3). We process 256 circles per iteration, until there are no circles left. We have to synchronize after the circleInBox operation. Some threads might be finished earlier than others, some might already try to work on a circle whose index is greater than the total number of circles in our image.

4.3 Shared memory inclusive scan

We now know which circles overlap with our section. Our current array of length 256 holds only zeros and ones. However, by performing a inclusive scan operation, not only will we know the total number of circles overlapping with our section, it will also become easier to store the final circle indices. Refer to the *sharedMemInclusiveScan()* step in figure 3. This step results in a array of length 256 which is simply the previous binary array after a inclusive scan operation.

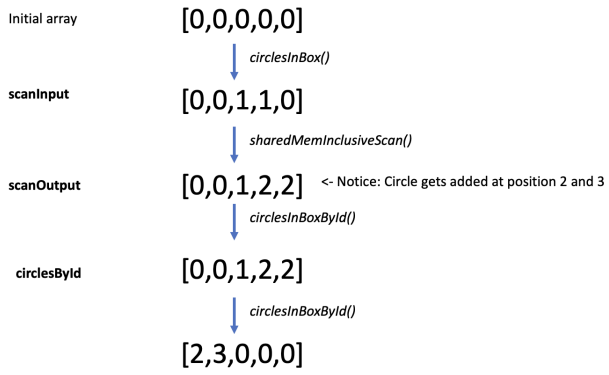

Figure 3: TODO

An exclusive scan operation was already provided by the assignment instructor. However, we do not want the first element to be zero in every case. For simplicity, a shared memory inclusive scan operation was added. It follows the same procedure as the exclusive scan operation, however, we do not subtract the element under the current index from the prefix sum up to that index.

After the inclusive scan operation, we synchronize the threads in our thread block once again.

We can easily utilize shared memory in this and the previous step. It was hinted at by the existing exclusive scan operation that shared memory can be used for this. Since every thread only needs to write a 0 or 1 in the scan input array and the arrays length is restricted to 256, this can reduce runtime and rule out unnecessary accesses to global memory. The same holds true for the actual scan operation.

```
// scanInput == binary array of length 256
// scanOutput == scanInput array after inclusive scan operation
for (int i = 0; i < cuConstRendererParams.numCircles; i += BLOCK_SIZE) {
        int circleIndex = threadIndex + i;
        circlesInBox(circleIndex, threadIndex, scanInput, boxL, boxR,
            boxT, boxB); // Find circles in box
        __syncthreads();
        sharedMemInclusiveScan(threadIndex, scanInput, scanOutput,
            scanScratch, BLOCK_SIZE); // Perform scan operation
        __syncthreads();
        // TODO: Get circle ids
        // TODO: Perform coloring
}
```

The last thing we need to do is to retrieve the actual indices of the circles overlapping with our section. We do this by calling the *circlesInBoxById()* function. This results in our final array holding the circle ids. See figure 3 **circlesById** for further explanation.

```
// scanInput == binary array of length 256
// scanOutput == scanInput array after inclusive scan operation
```

```
// circlesById == holds circle indices of circles overlapping with our
    section in correct order
for (int i = 0; i < cuConstRendererParams.numCircles; i += BLOCK_SIZE) {
        int circleIndex = threadIndex + i;
        circlesInBox(circleIndex, threadIndex, scanInput, boxL, boxR,
            boxT, boxB); // Find circles in box
        __syncthreads();
        sharedMemInclusiveScan(threadIndex, scanInput, scanOutput,
            scanScratch, BLOCK_SIZE); // Perform scan operation
        __syncthreads();
        if (threadIndex > 0) {
            circlesInBoxById(circleIndex, threadIndex, scanOutput,
                circlesById); // Find ids of circles in box
        } else if (scanOutput[threadIndex] == 1){
            circlesById[0] = circleIndex;
        }
        __syncthreads();
        // TODO: Perform coloring
}

__inline__ __device__ void
circlesInBoxById(uint circleIndex, uint threadIndex, uint* sOutput, uint*
    circleIndices) {
    int previousCircleCount = sOutput[threadIndex - 1];
    int currentCircleCount = sOutput[threadIndex];
    if (currentCircleCount == previousCircleCount + 1) { // A circle was
        added at this thread index -> add its circle Index
        circleIndices[sOutput[threadIndex - 1]] = circleIndex;
    }
}
```

In principle, we check the positions of our scan output array where the prefix sum increased by one in comparison to the previous index for each thread. If the prefix sum increased at a given thread index, we store the circle index for that thread index in the final output array.

We again have to synchronize our threads after this operation.

It is vital to mention that our circlesById array holds the circle indices in the **correct order of occurence**. Therefore, if circles 5, 2, and 13 would happen to overlap with our current section, the coloring operations would be executed in the correct ordering (2, 5, 13) due to this procedure.

Furthermore, since we operate on all circles in blocks of 256 (and these 256 circles are already in the correct order), the correct ordering is preserved for every iteration until there are no circles left.

This simplifies the coloring procedure immensely. Since each thread in our block performs coloring for one pixel independently, we do not have to worry about about atomicity any more with this implementation.

The final rendering procedure looks as follows:

```
__global__ void kernelRenderCircles() {
    // linear thread index
    int threadIndex = blockDim.y * threadIdx.y + threadIdx.x;

    // Treat each thread as a pixel later
    float4* imgPtr;
    float2 pixelCenterNorm;
    float4 color;
    int pixelX = blockIdx.x * blockDim.x + threadIdx.x;
    int pixelY = blockIdx.y * blockDim.y + threadIdx.y;

    // Define box for each threadblock needed for circleInBox() test (
        circleBoxTest.cu_inl)
```

```
float boxL = static_cast<float>(blockIdx.x) / static_cast<float>(
    gridDim.x);
float boxR = boxL + static_cast<float>(blockDim.x) / static_cast<
    float>(cuConstRendererParams.imageWidth);
float boxB = static_cast<float>(blockIdx.y) / static_cast<float>(
    gridDim.y);
float boxT = boxB + static_cast<float>(blockDim.y) / static_cast<
    float>(cuConstRendererParams.imageHeight);

// Define scan input and output arrays needed for inclusive scan
    operation (exclusiveScan.cu_inl)
// -> see REQUIREMENTS for further details
__shared__ uint scanInput[BLOCK_SIZE]; // binary array -> 1 if circle
     in box, 0 otherwise
__shared__ uint scanOutput[BLOCK_SIZE]; // holds output from scan
    operation
__shared__ uint scanScratch[2*BLOCK_SIZE];
__shared__ uint circlesById[BLOCK_SIZE]; // Holds final circle ids
    that are in box

float invWidth = 1.f / cuConstRendererParams.imageWidth;
float invHeight = 1.f / cuConstRendererParams.imageHeight;

imgPtr = (float4*) &cuConstRendererParams.imageData[4 * (pixelY *
    cuConstRendererParams.imageWidth + pixelX)];
color = *imgPtr;
pixelCenterNorm = make_float2(invWidth * (static_cast<float>(pixelX)
    + 0.5f), invHeight * (static_cast<float>(pixelY) + 0.5f));


// main procedure
for (int i = 0; i < cuConstRendererParams.numCircles; i += BLOCK_SIZE
    ) {
    int circleIndex = threadIndex + i;
    circlesInBox(circleIndex, threadIndex, scanInput, boxL, boxR,
        boxT, boxB); // Find circles in box
    __syncthreads();
    sharedMemInclusiveScan(threadIndex, scanInput, scanOutput,
        scanScratch, BLOCK_SIZE); // Perform scan operation
    __syncthreads();
    if (threadIndex > 0) {
        circlesInBoxById(circleIndex, threadIndex, scanOutput,
            circlesById); // Find ids of circles in box
    } else if (scanOutput[threadIndex] == 1){
        circlesById[0] = circleIndex;
    }
    __syncthreads();
    int numCirclesInBox = scanOutput[BLOCK_SIZE-1]; // last element
        of scan operation == number of circles in box

    // go over circles in box and update color
    for (int j = 0; j < numCirclesInBox; j++) {
        uint circleIndex = circlesById[j];
        float3 p = *(float3*)(&cuConstRendererParams.position[
            circleIndex * 3]);
        shadePixel(circleIndex, pixelCenterNorm, p, &color);
```

```
            }
            __syncthreads();
        }
        *imgPtr = color;
}
```

The shade pixel operation did not have to be adapted due to our previously established correct ordering for coloring.