Parallel Computing 2019/2020

CUDA Circle Renderer

Rendering of circular figures through NVidia CUDA parallelization

Andrea Spitaleri
spita90@gmail.com
andrea.spitaleri@stud.unifi.it

Abstract

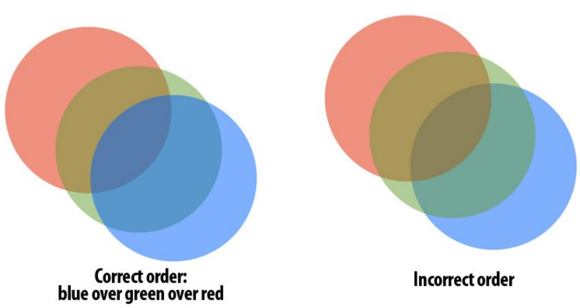
In this work we will discuss about two different approaches to rendering: Single-Thread CPU Sequential Computing and Multi-Thread GPU Parallel Computing.
In particular, we will test GPU against CPU for a task such as Rendering semi-transparent multicolored circles on the screen, an we will then evaluate the obtained results.

1. Overview

We want to draw colored circles on the screen. Circles can have different colors (coded as components of Red, Green, and Blue), dimension, and 3D position (horizontal position, vertical position, and depth).

Moreover, circles are semi-transparent. This latter fact adds a non-trivial level of difficulty in the process of rendering the final image.

Indeed, the color of a pixel is the result of a blending process done to the colors of all the circles that lie above that pixel. This blending function is non-commutative because, as in real life, the color resulting from a circle A in front of a circle B is different from that resulting from a circle B in front a circle A.



This difficulty lies in the fact that the final color of a pixel is given by 50% from the color of the circle in foreground, and by the other 50% from the result of blending the color of the circles behind the one in foreground, recursively. So there is a proper data dependency between circles with similar x,y coordinates.

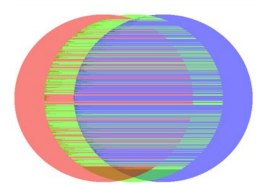
Actually, this is not a problem at all in case of Single-Thread rendering, because by simply rendering all circles in order from the farthest to the closest we gain the correct image. Conversely, in a Multi-Thread environment things get a little messy, because this data dependency makes parallelizing computation a little bit intricate.

2. The problem in Multi-Thread computation

To obtain a correct rendering of the final image, we must guarantee two invariants:

- Atomicity in Read-Edit-Write operations to the image pixel matrix, in order to avoid race conditions.
- Rendering order: for circles with similar x,y coordinates (so, for circles that overlap each other), render from the farthest to the nearest.

Not respecting these invariants will lead to a bad rendering, in terms of an incorrect image, as the one shown below.



3. The two different approaches

This sums up the Single-Thread algorithm:

```
Clear image
For each circle
    Compute circle's bounding box's coordinates
    For each pixel in the bounding box
        Compute pixel's continue coordinate (central
                         point – read Point- Sampling)
        If pixel's central point is inside the circle
            Compute circle color in that point
                Blend color with the one already in the pixel
```

In the CUDA Multi-Thread approach, the final image is divided in blocks of pixel, with a 1:1 association to block of threads. Rendering, in every single thread of each block looks something like this:

```
Clear image
For each circle in a segment of the whole array of
circles (so that each block analyzes the whole array
of circles)
    If the circle's bounding box has at least one point
    inside the current block
        Save its index in shared memory (shared
        between threads of the same block)
Now, for each block there's a list of indexes of circles
bounding boxes that have at least one point lying in the
current block
For each circle associated with this block
    If the circle has actually at least one point lying in
    current block
        Compute circle color in that point
        Blend the color with the one already
        in the pixel
```

The two invariants, in the Multi-Thread algorithm are guaranteed because:

- We have no contention on pixels, because each thread deals just with its only single pixel.
- Circles are passed to the renderer already in depth order, and the process of reorganization circle indexes in lists for each block is done without breaking this invariant.

4. Software implementation

My work is available at the following URL: https://github.com/spita90/CUDA-Circle-Renderer and was made with Qt Creator in C++ and CUDA-C languages.
If you launch the program without specifying command-line options, you will get a list of possible options:

-r <cpu/cuda>	Allows to select the renderer you want to use to render the final image
-s [size]	Allows to specify, in pixels, the side of the square final image (default is 1024 so the image will be 1024x1024)
-b	Starts in benchmark mode, where the rendering of the same circles is done 5 times by both the CPU and the GPU
-m <good/bad>	Allows to choose which implementation of the parallel rendering algorithm to use: the correct one or the one subject to race conditions

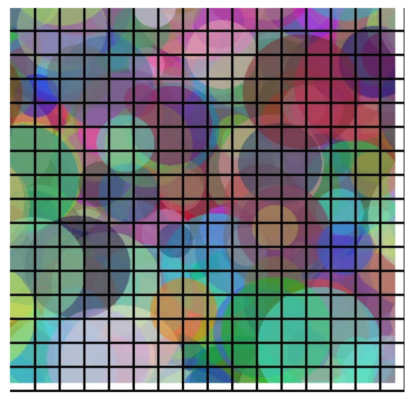
5. CUDA rendering dynamics

Now we will take a look at some important pieces of code, the ones that actually "do the magic".
Most of that magic is in the *cudaRenderer.cu* file:

To each block of threads, of size *ThrInBlock_X* times *ThrInBlock_Y,* is associated a squared portion of the final image of same size (in pixels, obviously).

```
short blockLeftIdx =
blockIdx.x * ThrInBlock_X;
short blockRightIdx =
blockLeftIdx + ThrInBlock_X - 1;
short blockTopIdx =
blockIdx.y * ThrInBlock_Y;
short blockBottomIdx =
blockTopIdx + ThrInBlock_Y - 1;
```

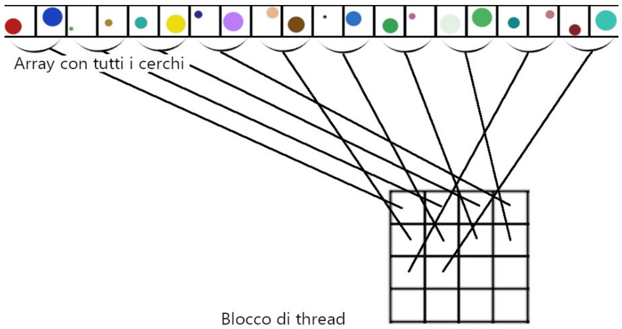
Final image divided in blocks



Each thread in a block analyzes a portion of the whole circles array (so that each block analyzes the whole array) to evaluate if the circle's bounding box has at least one point in the current block. If that's the case, it saves the circle's index inside a little local array.

```
int circlesPerThread =
    (cuConstRendererParams.numCircles +
    ThrInBlock - 1) / ThrInBlock;
int circleStartIdx = threadIndex * circlesPerThread;
int circleEndIdx = circleStartIdx + circlesPerThread;
// last thread takes all the remaining circles
if (threadIndex == ThrInBlock) {
    circleEndIdx = cuConstRendererParams.numCircles;
}
```

The `numCircles + ThrInBlock – 1` allows to compensate the truncation to integer, and allows the last thread non to be loaded with too much circles, so analysis times are contained and we don't have waste of compute cicles for the other threads.




```
int threadCircleCount = 0;
int threadCircleList[CirclesPerThread];
for (int c = circleStartIdx; c<circleEndIdx; c++){
  if (c >= cuConstRendererParams.numCircles)
    break;
  float3 position =
  *(float3*)(&cuConstRendererParams.position[c * 3]);
  float radius = cuConstRendererParams.radius[c];
  if (circleInBoxConservative(position.x, position.y,
    radius, blockLeftNormIdx, blockRightNormIdx,
    blockBottomNormIdx, blockTopNormIdx) == 1){
    threadCircleList[threadCircleCount++] = c;
  }
}
```

So, in this part we always check not to go over circle array's border, and then we use the function `circleInBoxConservative`, supplied with the initial material, to check if the circle's bounding box has at least a point inside current block. If that's the case it saves the circle's index inside a little local array.
Now `threadCircleCount` contains, of the portion of analyzed circles, how many of them lie in the current block, and in `threadCircleList`, of the portion of analyzed circles, the indexes of that circles. These partial results are then loaded in an array shared between all threads of current block.

```
__shared__ uint circleCount[ThrInBlock];
__shared__ uint circleList[MaxCircles];
__shared__ uint indexList[ThrInBlock];

circleCount[threadIndex] = threadCircleCount;
__syncthreads();
sharedMemExclusiveScan(threadIndex, circleCount,
indexList, circleList, ThrInBlock);
__syncthreads();
uint privateIndex = indexList[threadIndex];
for(int i=0; i<threadCircleCount; i++){
  circleList[privateIndex++] = threadCircleList[i];
}
__syncthreads();
```

After the first `__syncthreads()`, `circleCount` has become the sequence containing the number of circles belonging to current block found by each thread in their respective portion of array of circles.
Then we use the function `sharedMemExclusiveScan`, supplied with the initial material, to compute the series of the sequence `circleCount` and to put that series in `indexList`, so we are then able to know, for each thread, the starting position (`privateIndex`) from which each thread can copy the indexes of the circles he found, in `circleList`.
The series does not include the last sum (it is exclusive), so to compute the total number of circles in the current block we add the last sum to the last value of the series.

```
int totalCircles =
indexList[ThrInBlock-1] + circleCount[ThrInBlock-1];
```

Now that we have the list of all indexes of circles lying on current block we can start the drawing part of the rendering.
We compute the continue coordinate of the center of the pixel, sampling the color of the pixel based on the fact if the circle actually lies on that pixel center or if not. This tecnique, in GCI slang, is called Point-Sampling.

```
int pixelXCoord = blockLeftIdx + threadIdx.x;
int pixelYCoord = blockTopIdx + threadIdx.y;
float2 pixelCenterNorm = make_float2(
  invWidth*(static_cast<float>(pixelXCoord)+0.5f),
  invHeight*(static_cast<float>(pixelYCoord)+0.5f));

float4* imgPtr =
(float4*)(&cuConstRendererParams.imageData[4*
(pixelYCoord * imageWidth + pixelXCoord)]);

float4 pixelData = *imgPtr;
```

Now each thread renders a pixel cycling on all circles on current block, starting from the farthest one. This invariant is guaranteed by the fact that circles are supplied to the renderer already in depth order, and are maintained in that order during the copy of their index in `circleList`.
Than we check if the circle is actually on the pixel, and, if true, we compute the new pixel color and save it on the shared memory.

```
for (int i=0; i<totalCircles; i++){

  int circleIndex = circleList[i];
  float3 position =*(float3*
  (&cuConstRendererParams.position[circleIndex * 3]);
  float radius =
  cuConstRendererParams.radius[circleIndex];
  float diffX = position.x - pixelCenterNorm.x;
  float diffY = position.y - pixelCenterNorm.y;
  float pixelDist = diffX * diffX + diffY * diffY;
  float maxDist = radius * radius;
```

```
    if (pixelDist <= maxDist){
      int index3 = 3 * circleIndex;
      float3 rgb = *(float3*)
        &(cuConstRendererParams.color[index3]);
      float alpha = .5f;
      float oneMinusAlpha = 1.f - alpha;
      pixelData.x = alpha * rgb.x +
        oneMinusAlpha * pixelData.x;
      pixelData.y = alpha * rgb.y +
        oneMinusAlpha * pixelData.y;
      pixelData.z = alpha * rgb.z +
        oneMinusAlpha * pixelData.z;
      pixelData.w = alpha +
        pixelData.w;
    }
  }
  *imgPtr = pixelData;
```

In order to reduce access to global memory to minimum, all immutable data are copied in constant memory before computation. We have access to global memory only when reading the pixel matrix and when overwriting it.

6. Rendering performance analysis

Using the integrated benchmark function, which produces for 5 times the rendering of the same circles with both CPU and GPU, and then compares the mean computation times, we can look at the speedups that we can obtain by varying some parameters such as circles number, resolution of the final image, and size of blocks of threads.

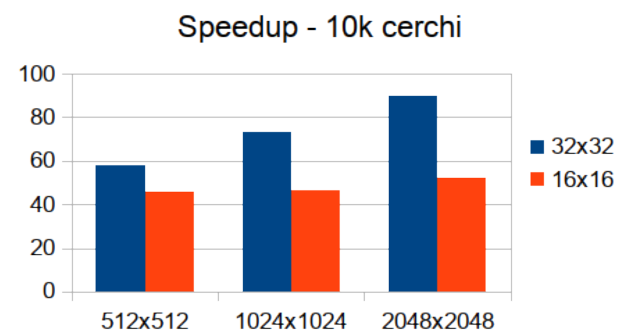
The following benchmarks are done with this system:

Laptop Xiaomi Mi Notebook Pro
CPU Intel Core i7-8550U (8 logic cores @ 1.8 - till 4 GHz in Turbo Boost)
RAM 16 GB DDR4
O.S. Manjaro Linux 18.1.4 (based on Arch Linux) with Linux Kernel 5.4
NVidia GeForce MX150
3 Streaming Multiprocessors Global memory: 2 GB
Compute Capability: 6.1 Shared mem. per block: 48KB
CUDA Libraries 10.1

To be able to use the graphic chip only when needed I had to install bumblebee and optirun packages, which try to port NVidia Optimus technology to the Linux open source environment. Since these are not official NVidia packages, the performance obtained from the graphic chip may not be at its maximum.

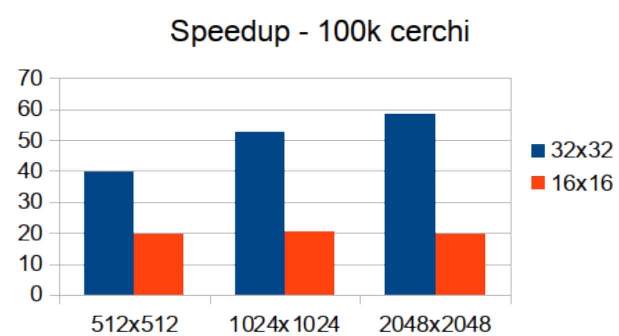
With 10k circles we obtained the following mean times (in milliseconds):

	Ris. 512x512 Blocchi 32x32	Ris. 512x512 Blocchi 16x16	Ris. 1024x1024 Blocchi 32x32	Ris. 1024x1024 Blocchi 16x16	Ris. 2048x2048 Blocchi 32x32	Ris. 2048x2048 Blocchi 16x16
CPU	488	490	1827	1840	7050	7046
GPU	8	11	25	40	78	135
Speedup	57,9	46,1	73,3	46,5	90,1	52,1



With 100k circles we obtained the following mean times (in milliseconds):

	Ris. 512x512 Blocchi 32x32	Ris. 512x512 Blocchi 16x16	Ris. 1024x1024 Blocchi 32x32	Ris. 1024x1024 Blocchi 16x16	Ris. 2048x2048 Blocchi 32x32	Ris. 2048x2048 Blocchi 16x16
CPU	4464	4426	17812	17819	69951	69906
GPU	112	226	338	872	1198	3544
Speedup	39,7	19,5	52,6	20,4	58,3	19,7



Comparing both speedup graphs we can immediately observe that with blocks of 32x32 threads we have much better performance than blocks of 16x16.
We can also note that, in case of blocks of 32x32 pixels, as we increase output image resolution, speedup tends to increase as well, regardless of the number of circles.
In the test with 10k circles, with output image resolution 2048x2048 we even get a staggering 90x speedup factor! This highlights the fact that GPUs can be very helpful for tasks such as rendering figures on screen.

Andrea Spitaleri