Assignment 2

- Qiaoyu Deng (qdeng@andrew.cmu.edu)
- Changkai Zhou (zchangka@andrew.cmu.edu)

Part 1: SAXPY (5 pts)

1) Results on ghc31 (with GTX 1080)

	Time (ms)	Bandwidth (GB/s)
CPU-based	21.1	14.1
GPU without memcpy	1.1	211.8
GPU with memcpy	25.4	8.8

Low arithmetic intensity causes the major difference of time for GPU between the one without *memcpy* and the one with *memcpy*. PCIe-x16 bus is the bottleneck of the runtime due to relatively high cost from main memory access.

2) Memory bandwidth is 320 GB/s according to the specification, which is almost consistent with the estimated bandwidth (211.8 GB/s).

The bandwidth of PCIe-x16 3.0 is 32 GB/s according to this article, which is almost consistent with the estimated bandwidth (14.1 GB/s).

Part 2: Parallel Prefix-Sum (10 pts)

Results on ghc29 (with GTX 1080): more detials on source code.

- Exclusive scan

Element Count	Fast Time	Your Time	Score
10000	0.238	0.284	1.25
100000	0.336	0.339	1.25
1000000	1.066	0.787	1.25
2000000	1.626	0.825	1.25
		Total score:	5/5

- Find repeat

Element Count	Fast Time	Your Time	Score
10000	0.277	0.085	1.25
100000	0.675	0.596	1.25
1000000	1.693	0.957	1.25
2000000	2.361	1.068	1.25
		Total score:	5/5

Part 3: A Simple Circle Renderer

Design 1

Final Results on [ghc27]

Scene Name	Fast Time (Tf)	Your Time (T)	Score
rgb	0.2379	0.2249	12
rand10k	3.3355	3.6144	12
rand100k	28.8924	30.2437	12
pattern	0.3466	0.6665	8
snowsingle	15.1902	13.2563	12
big little	23.6971	24.1122	12
		Total score:	68/72

Decomposition

We apply **Box-Level** and **Circle-level Parallelism** in different main steps to optimization the problem solving. The problem is decompose into these steps below:

1. Partitioning

Equally partition the image into numerous square boxes with same number of pixels in each box.

2. Scaning

Based on each box, we use one thread on each box to find the circles that is intersected with it. and return a `circleList`. Inside `circleList` there are circles' indexes put at the first *numCircleInBox* positions. Here we use **Box-Level Parallelism**.

We let the box side length be 32 pixels, so for a 1024 * 1024 image, we will have 32 * 32 boxes. And each box is checking its intersected circles at the same when kernel starts. Inside the kernel, we use shared memory to store 1024 circles ((float3 rgb + float3 position + float radius) * 1024 = 28KB < 48KB) for every iteration until all of circles has been checked for every box. And we use 16 * 16 boxes inside a block, since 256 threads in a sm is a very good number since 128 threads can work at the same time, while another 128 threads can wait in the memory when some warps is stalled for memory accessing.

3. Rendering

Render each pixel with the circles that cover the pixel by iterating only the circles in the box where the pixel locates. Here, we use **Pixel-Level Parallelism** and treat each pixel as a thread.

Since now we have what circle the current box is intersected with, then we can iterate every pixel within that box to see if they are in the range of circles, if it is true, we then render that pixel. And the index of circles has also indicate order of redenering, so we do not need to worry about order. For this part, we start another kernel function called `kernelRenderPixel` to render pixel, and we assign every pixel to every threads, so with many piexls need to be redenerd, we assign *32 * 32 = 1024* threads to every thread, so the block in `kernelRenderPixel` is completely equal to the box in `kernelCircle` , and then we can use share memory to loads *32 * 32 = 1024 pixels* and some part of circles in *circleList* before we start. Since max shared memory that can be used by each block is 48KB, for *1024* pixels, we will use 16KB(float4 imageData * 1024 pixels), and the rest 32KB can be used to load 1024 circles ((float3 rgb + float3 position + float radius) * 1024 = 28KB), when all pixels iterate the first 1024 circles over, we can load another 1024 circles until all the circles that intersected with current block(or previous box) is completed.

Synchronization

No explicit synchronization thanks to the independence of workload in every parallel computing process.

However, we use synchronization every time when we load data into share memory and we synchronize once we finish scanning all of circles in the share memory when we are ready for the next stage of loading circles into share memory both in **Scaning** and **Redenering**.

What's more, the implicit synchronization occurs at the ending point of **Scaning** and **rendering** with the function `cudaDeviceSynchronize()` to wait for going ahead on CPU until all kernels launched on CPU have finished their tasks.

Reducing Communication

- Synchronization

  Since we give each box a unique circle list when iterating circles in **Box-level Parallelism**, we can avoid using mutex to make all of boxes access circle list at the same time which will decrease the performance a lot.

And for **Rendering** process in **Pixel-Level Parallelism**, since each pixel can know which box it located, so they can have a circle list that the might be in the range of circle. So every pixel can have their own circle list which is far lees than `numCircles` and iterate to redene color.

So we use pixel to avoid synchronization.

- Main Memory Access

  For every load and store insctruction with the cuda program, we use share memory to increase the performance. For example, we will load 1024 circles for every block, and then all of threads(pixels) can access that region with a higher rate in **Scaning** process. For **Rendenering** process, since we load all of pixels's information into shared memory, we only need load 1024 * 1024 pixels for one time, and then write them back in the end, for circles it is also true, each block will only load circles in `circleList` only one time. So we reduce so many global memory communication.

Design 2

This design is worse than design 1, but we actually expect a better performance on design 2 and spend more time on it.

Scene Name	Fast Time (Tf)	Your Time (T)	Score
rgb	0.2398	0.7119	6
rand10k	3.3392	4.9718	9
rand100k	29.7127	37.1238	11
pattern	0.3461	1.7113	5
snowsingle	15.2469	16.3425	12
big little	23.1879	28.6292	11
		Total score:	54/72

Decomposition

We apply both **Pixel-Level** and **Circle-Cevel Parallelism** in different steps to optimization the problem solving. The problem is decompose into these steps below:

1. Partitioning

Equally partition the image into numerous square boxes with same number of pixels in each box.

2. Assignment

Assign each box with the indexes of circles intersecting with the box. Here we use the implementation of **Circle-Level Parallelism**. One circle is treated as a thread and planned to **iterate all boxes that the circle intersects** by updating the circle index array of the boxes. The circle index array in each box has the length of the total number of circles and the value of the ***i***-th index signifies whether the *i*-th circle and the box are intersecting (0-No, 1-Yes).

We let the box side length be *32* pixels, so for a *1024 * 1024* image, we will have *32 * 32* boxes. And each circle in the `kernelCircle` will compute independently whether it has insected with those boxed and store result in the array. Since every item whoes index in the array represent whether current index circle is intersected with that box, all the circles can operate the array at the same without mutex. For `kernelCircle` , we assign every block for *1024* circles since the number of circles is unbounded, so we need to assign circles as much as possible to each block.

3. Scanning

Use **exclusive_scan** method to scan each index array in each box and compress them **into minimum space**.

We take a simple example:

if we get *0011 1001 1000 0001* as the output from the `kernelCircle` which means the 0th box has a intersection list: *0011*, since *2nd a*nd *3rd* position is *1*, it means that circles with index number 2 and 3 are intersected with box with index 0. For the same reason, *1001* means that box *1* is intersected with circles *0* and *1*, *1000* means that box *2* is intersected with circles *0*, and *0001* means box 3 is only intersected with circle *3*. But if number of circles is very large, we will waste too much time on iterating list. So we can first use **exclusive_scan** provided by thrust to preprocess. After that we will get: *0001 2333 4555 5555*, now we can do something similar to `find_repeats` check whether the *ith* number is different with *(i + 1)th* number, if it is true, we think this *ith* circle has intersection with current box, and we store this *i* to a new array called `circleList` , the index in `circleList` is indicated by current locations in scan result minus the first value of every *numCircles* circles. So for the output *0001*, the compressed circleList is *3*, and for those last circle also intersected we add a more intger to indicated the total number of circles in current box, so *1001* 's circleList is *03(-1)(-1)2* The last *2* represents there are 2 circles intersted with box 1. So the fina compressed list should be: **23(-1)(-1)[2] 03(-1)(-1)[2] 0(-1)(-1)(-1)[1] 3(-1)(-1)(-1)[1]** now, we can have a `circleList` that the first n index before *-1* occurs is all the circles that a box intersected.

4. Rendering

Render each pixel with the circles that cover the pixel by iterating only the circles in the box where the pixel locates. Here, we use **Pixel-Level Parallelism** and treat each pixel as a thread.

Since now we have what circle the current box is intersected with, then we can iterate every pixel within that box to see if they are in the range of circles, if it is true, we then render that pixel. And the index of circles has also indicate order of redenering, so we do not need to worry about order. For this part, we start another kernel function called `kernelRenderPixel` to render pixel, and we assign every pixel to every threads, so with many piexls need to be redenerd, we assign *32 * 32 = 1024* threads to every thread, so the block in `kernelRenderPixel` is completely equal to the box in

`kernelCircle` , and then we can use share memory to loads *32 * 32 = 1024 pixels* and some part of circles in *circleList* before we start. Since max shared memory that can be used by each block is 48KB, for *1024* pixels, we will use 16KB(float4 imageData * 1024 pixels), and the rest 32KB can be used to load 1024 circles ((float3 rgb + float3 position + float radius) * 1024 = 28KB), when all pixels iterate the first 1024 circles over, we can load another 1024 circles until all the circles that intersected with current block(or previous box) is completed.

Synchronization

No explicit synchronization thanks to the independence of workload in every parallel computing process.

However, the implicit synchronization occurs at the ending point of **assignment** and **rendering** with the function `cudaDeviceSynchronize()` to wait for going ahead on CPU until all kernels launched on CPU have finished their tasks.

Reducing Communication

- Synchronization

During the step of **assignment**, there are two main options of index storation based on **Circle-Level Parallelism**. In each box, if we store the original indexes of circles (eg. indexes = [3, 6, 210, 10323]), it will trigger the conflicts among parallel threads as circles and cause a high cost of synchronization. Therefore, we use the *num(circles)*-length array of 0/1 signals to store indexes of circles, which sacrifices the space for the time by cutting down the synchronization.

Another simple point is to use **Pixel-Level Parallelism** rather than **Circle(Or Box)-Level Parallelism** during **rendering** step. It definitely saves overhead.

- Main Memory Access

Obviously, partitioning the image into many boxes helps avoid that every pixel goes through the whole list of circles, which is an unaffordable cost as for main memory access.

Besides, in the step of **assignment**, it greatly cuts down the overhead caused by main memory access by using **Circle-Level Parallelism** instead of **Box-Level Parallelism**. If using **Box-Level Parallelsim**, the main memory will be accessed by *num(circles) * num(boxes)* times. However, using **Circle-Level Parallelism** helps cut it down to *num(circles) * mean(num(boxes per circle intersecting))* times.

In addition, during the **scanning** step, we copy the index array from main memory into the block shared memory to save overhead.

For **redenering** process, since we load all of pixels's information into shared memory, we only need load 1024 * 1024 pixels for one time, and then write them back in the end, for circles it is also true, each block will only load circles in `circleList` only one time. So we reduce so many global memory communication.

Other approaches

V1.0

Most intuitive: directly use pixel-level parallelism by iterating all circles to render. Well, definitely a poor idea.

V2.0

Box-Level Parallelism + Pixel-level Parallelism

Similar decomposition of problem, but use **Box-Level Parallelism** rather than **Circle-Level Parallelism** during the **assignment** step. As is mentioned above, it costs more overhead because of more main memory access.

V2.1

Box-Level Parallelism + Pixel-level Parallelism + Hierarchical Structure

Optimized version of V2.0. During the **assignment** step, use hierarchical assignments and iteratively decrease the size of boxes for several times. In theory, it will decrease the total access of main memory. However, the result has no obvious improvement.

V2.2

Box-Level Parallelism + Pixel-level Parallelism + shared memory Optimization

The best performance is reached by design 1 intrduced above.

V3.0

Circle-Level Parallelism + Pixel-level Parallelism

The idea introduced above without the delicate optimization for main memory access.

V3.1

Circle-Level Parallelism + Pixel-level Parallelism + More Optimization

The design 2 introduced above.

Conclusion

Sometimes, simplest wins. We design a complicated one with far more steps process but end with worse performance, maybe we add more complicated steps increasing its running time. And for lock, we think it is better not using it to ensure mutual exclusive property, because it will make parallel into sequencial process. And the most imporatnt things to do to improve the performance is trying to use share memory as much as possible, and reducing the refrence to global memory. Also, setting parameters is also important, since we need to change them according to the hardware specification.