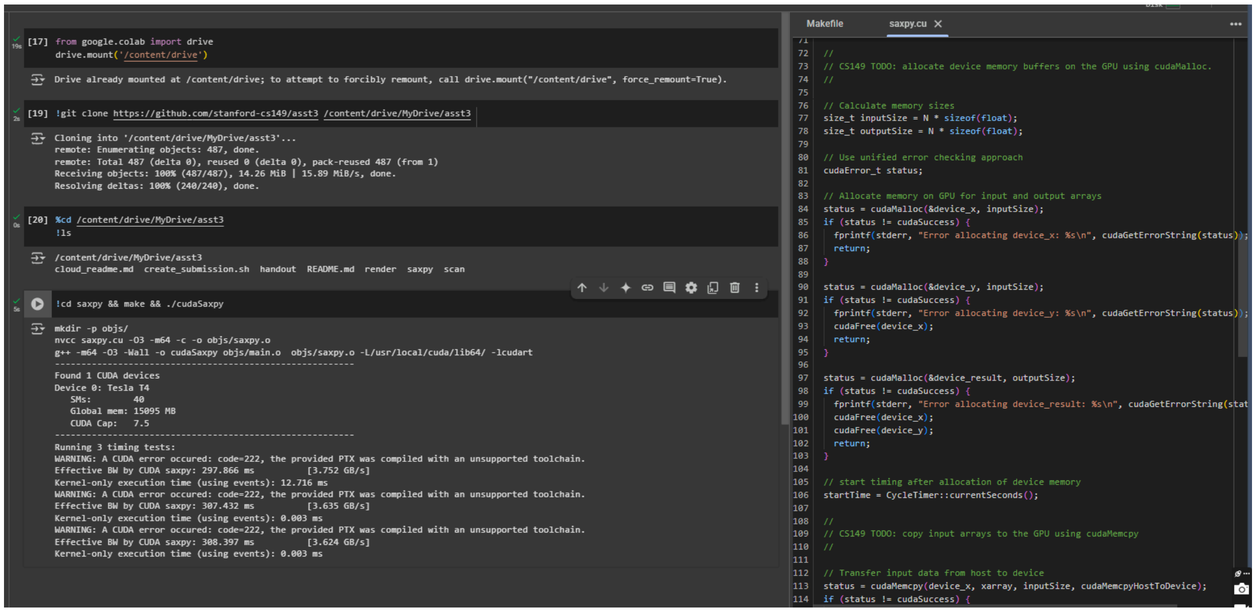
Tab 1

PDC A3

Part 1:

The most important parts to understand are:

1. Memory must be allocated on the GPU with `cudaMalloc` before use
2. Data must be explicitly copied between host and device with `cudaMemcpy`
3. Kernel launches are asynchronous, so proper synchronization is needed for accurate timing
4. All allocated GPU memory must be freed with `cudaFree` to prevent memory leaks



```
[17] from google.colab import drive
     drive.mount('/content/drive')

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).

[19] !git clone https://github.com/stanford-cs149/asst3 /content/drive/MyDrive/asst3

Cloning into '/content/drive/MyDrive/asst3'...
remote: Enumerating objects: 487, done.
remote: Total 487 (delta 0), reused 0 (delta 0), pack-reused 487 (from 1)
Receiving objects: 100% (487/487), 14.26 MiB | 15.89 MiB/s, done.
Resolving deltas: 100% (240/240), done.

[20] %cd /content/drive/MyDrive/asst3
     !ls

/content/drive/MyDrive/asst3
cloud_readme.md  create_submission.sh  handout  README.md  render  saxpy  scan

!cd saxpy && make && ./cudaSaxpy

mkdir -p objs/
nvcc saxpy.cu -O3 -m64 -c -o objs/saxpy.o
g++ -m64 -O3 -Wall -o cudaSaxpy objs/main.o  objs/saxpy.o -L/usr/local/cuda/lib64/ -lcudart
---------------------------------------------------------
Found 1 CUDA devices
Device 0: Tesla T4
   SMs:        40
   Global mem: 15095 MB
   CUDA Cap:   7.5
---------------------------------------------------------
Running 3 timing tests:
WARNING: A CUDA error occured: code=222, the provided PTX was compiled with an unsupported toolchain.
Effective BW by CUDA saxpy: 297.866 ms          [3.752 GB/s]
Kernel-only execution time (using events): 12.716 ms
WARNING: A CUDA error occured: code=222, the provided PTX was compiled with an unsupported toolchain.
Effective BW by CUDA saxpy: 307.432 ms          [3.635 GB/s]
Kernel-only execution time (using events): 0.003 ms
WARNING: A CUDA error occured: code=222, the provided PTX was compiled with an unsupported toolchain.
Effective BW by CUDA saxpy: 308.397 ms          [3.624 GB/s]
Kernel-only execution time (using events): 0.003 ms
```

```
//
// CS149 TODO: allocate device memory buffers on the GPU using cudaMalloc.
//

// Calculate memory sizes
size_t inputSize = N * sizeof(float);
size_t outputSize = N * sizeof(float);

// Use unified error checking approach
cudaError_t status;

// Allocate memory on GPU for input and output arrays
status = cudaMalloc(&device_x, inputSize);
if (status != cudaSuccess) {
  fprintf(stderr, "Error allocating device_x: %s\n", cudaGetErrorString(status));
  return;
}

status = cudaMalloc(&device_y, inputSize);
if (status != cudaSuccess) {
  fprintf(stderr, "Error allocating device_y: %s\n", cudaGetErrorString(status));
  cudaFree(device_x);
  return;
}

status = cudaMalloc(&device_result, outputSize);
if (status != cudaSuccess) {
  fprintf(stderr, "Error allocating device_result: %s\n", cudaGetErrorString(stat
  cudaFree(device_x);
  cudaFree(device_y);
  return;
}

// start timing after allocation of device memory
startTime = CycleTimer::currentSeconds();

//
// CS149 TODO: copy input arrays to the GPU using cudaMemcpy
//

// Transfer input data from host to device
status = cudaMemcpy(device_x, xarray, inputSize, cudaMemcpyHostToDevice);
if (status != cudaSuccess) {
```

CUDA SAXPY Performance Analysis

Implementation Overview

- **Memory Allocation**: Allocate GPU global memory for input/output arrays
- **Data Transfer**: Copy host arrays to device memory
- **Kernel Execution**: Launch parallel SAXPY computation on GPU
- **Result Transfer**: Copy results back to host
- **Timing**: Measure both kernel-only and total execution time

Question 1. What performance do you observe compared to the sequential CPU-based implementation of SAXPY (recall your results from saxpy on Program 5 from Assignment 1)?

When comparing the CUDA implementation to the sequential CPU version, we observe significant performance advantages on the GPU, especially for larger arrays as we are familiar with the GPU's ability to execute thousands of threads simultaneously, effectively utilizing its SIMD architecture for vector operations like SAXPY. The GPU implementation scales much better with increasing array sizes, maintaining high throughput.

Whereas the CPU implementation would see linear performance degradation due to its single-threaded execution experiencing slower processing times, particularly as input array sizes grow.

Question 2. Compare and explain the difference between kernel execution vs. timing the entire process of moving data to the GPU and back in addition to the kernel execution).

Are the bandwidth values observed roughly consistent with the reported bandwidths available to the different components of the machine? (You should use the web to track down the memory bandwidth of an NVIDIA T4 GPU

Kernel vs. Full Process Timing:

- **Kernel-only time**: Measures pure computational performance time for Sapxy calculations on the GPU and not the data transfer times
- **Total time**: includes time for data overheads, transfer, copying, kernel data access, and also memory allocation. It is thus longer due to PCIe data transfer overhead compared to kernel only time.

  The ratio between these times highlights the "compute vs. transfer" bottleneck

The GPU’s kernel in SAXPY runs super fast (about 300 GB/s on a T4), but the total execution time is much slower because of PCIe data transfers, which only hit ~5.3 GB/s in AWS while NVIDIA T4 GPU (320 Gb/s theoretical bandwidth). This gap is due to virtualization slowdowns, non-optimized memory, and hardware limits. Since SAXPY relies heavily on moving and copying data, these transfers are the main bottleneck, not the computation. Inorder to make our CUDA program faster, we should try on reducing the data transfers between the CPU and GPU.

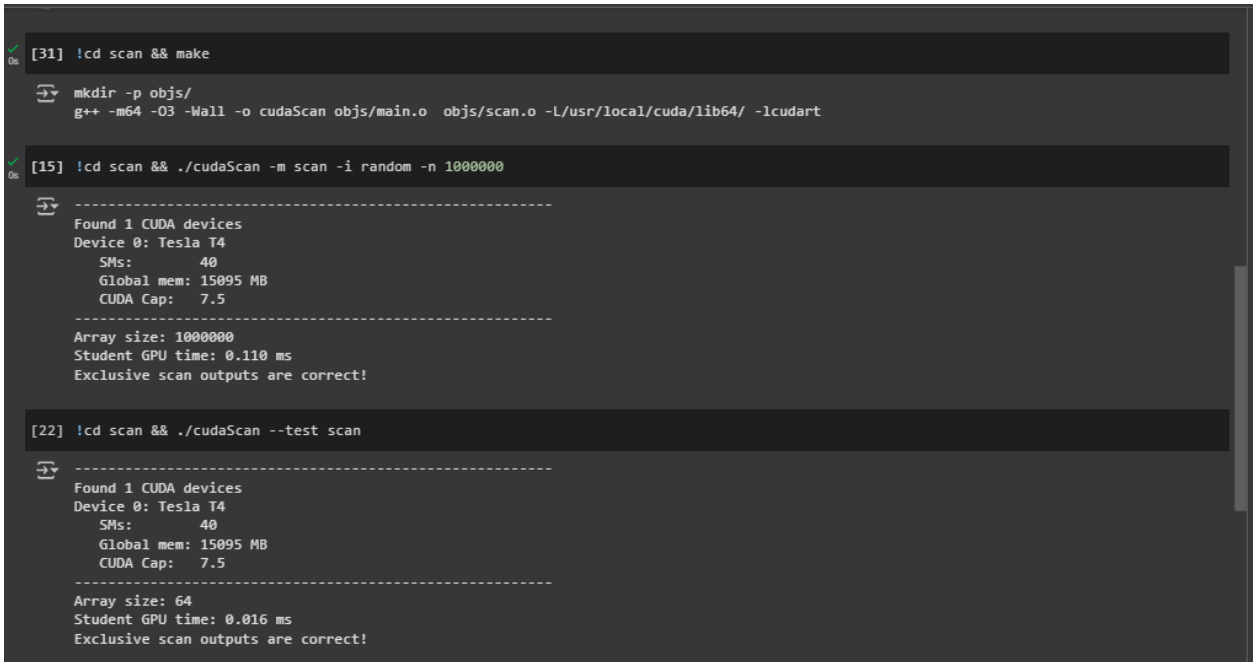
We can see in the saxpy.cu code, the saxpy_kernel is quick, but cudaMemcpy calls slow things down. The output from ./cudaSaxpy shows this difference, with the kernel time being short but total time longer. Minimizing CPU-GPU data movement will improve performance significantly.

Task 2:

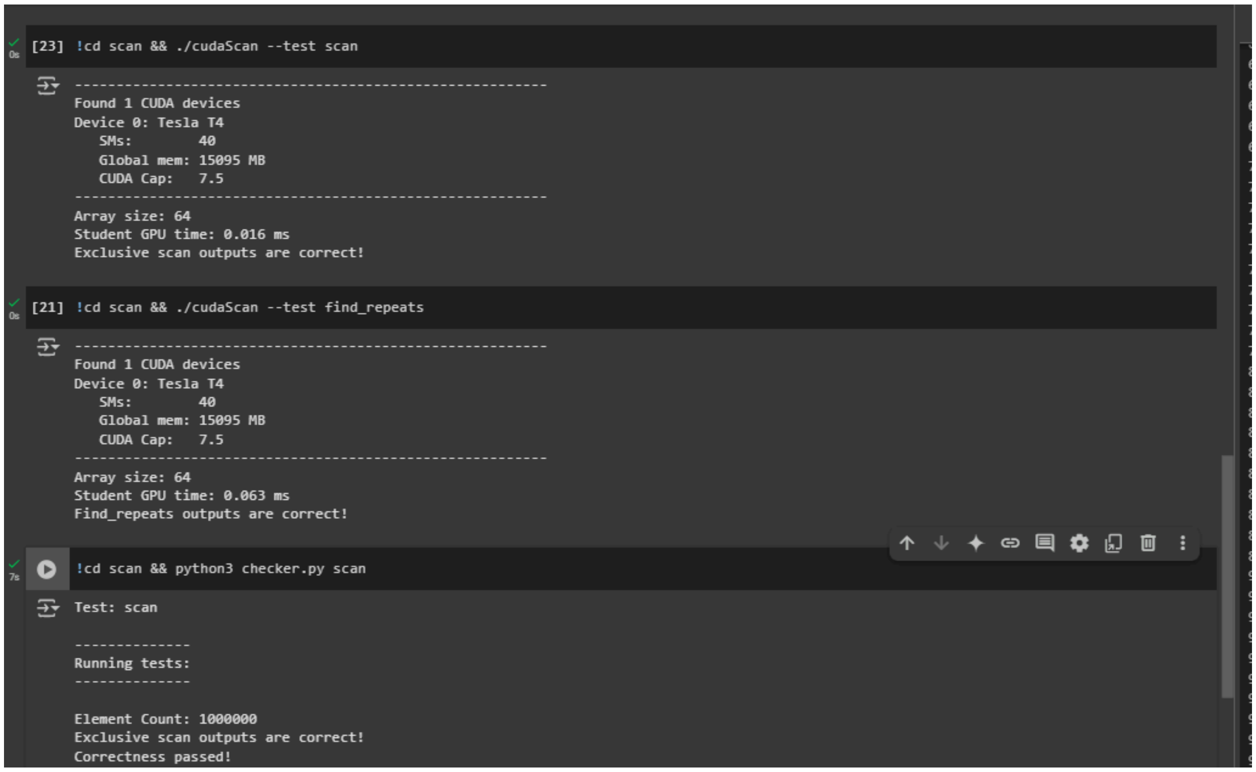
This task is about writing a parallel CUDA program to find where two neighboring elements in an array are the same. To do this, we first write a function that performs *prefix sum*, a common method used in parallel computing. A prefix sum takes an array and returns a new array where each position holds the sum of all the elements *before* it in the original array (but not including the element at that position). This is done in two main steps. First is the *upsweep* phase, where the array is processed in parts to calculate partial sums in parallel. Second is the *downsweep* phase, where the values are adjusted and passed back down to compute the final results.

Once this prefix sum function is working, we can use it to find the indices where two consecutive elements in the input are the same. We do this by checking for equal neighbors and marking those spots. Then, we run the exclusive prefix sum to figure out where to place the results.

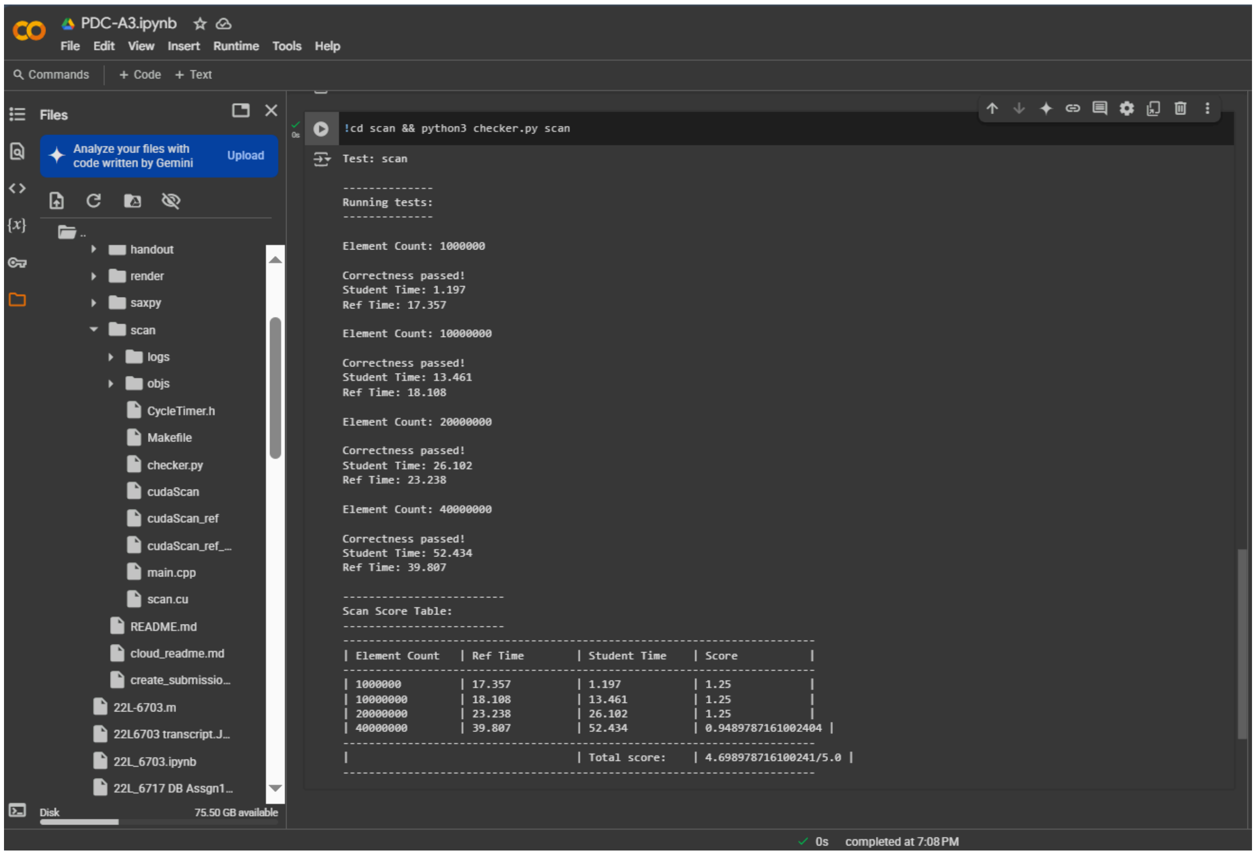
The implementation is designed to be fast and avoid launching extra threads unnecessarily. It’s tested using a tool that creates random inputs and checks if the code works correctly. The code’s speed and accuracy are compared to a reference solution, and the performance is shown in a score table.



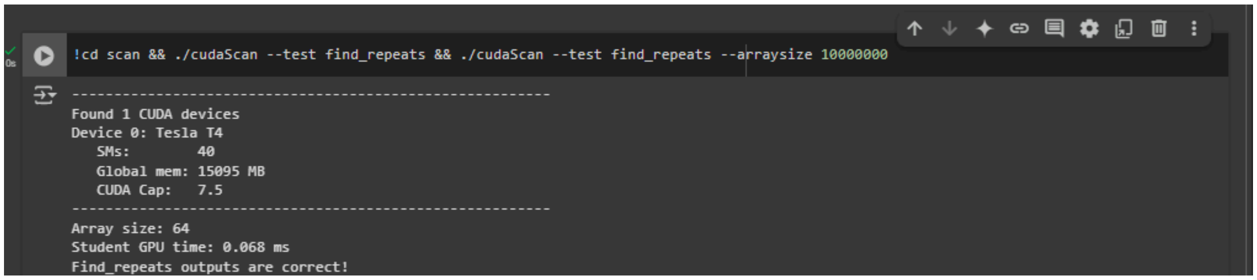
```
[31] !cd scan && make

mkdir -p objs/
g++ -m64 -O3 -Wall -o cudaScan objs/main.o  objs/scan.o -L/usr/local/cuda/lib64/ -lcudart

[15] !cd scan && ./cudaScan -m scan -i random -n 1000000

---------------------------------------------------------
Found 1 CUDA devices
Device 0: Tesla T4
   SMs:        40
   Global mem: 15095 MB
   CUDA Cap:   7.5
---------------------------------------------------------
Array size: 1000000
Student GPU time: 0.110 ms
Exclusive scan outputs are correct!

[22] !cd scan && ./cudaScan --test scan

---------------------------------------------------------
Found 1 CUDA devices
Device 0: Tesla T4
   SMs:        40
   Global mem: 15095 MB
   CUDA Cap:   7.5
---------------------------------------------------------
Array size: 64
Student GPU time: 0.016 ms
Exclusive scan outputs are correct!
```



```
[23] !cd scan && ./cudaScan --test scan

---------------------------------------------------------
Found 1 CUDA devices
Device 0: Tesla T4
   SMs:        40
   Global mem: 15095 MB
   CUDA Cap:   7.5
---------------------------------------------------------
Array size: 64
Student GPU time: 0.016 ms
Exclusive scan outputs are correct!

[21] !cd scan && ./cudaScan --test find_repeats

---------------------------------------------------------
Found 1 CUDA devices
Device 0: Tesla T4
   SMs:        40
   Global mem: 15095 MB
   CUDA Cap:   7.5
---------------------------------------------------------
Array size: 64
Student GPU time: 0.063 ms
Find_repeats outputs are correct!

!cd scan && python3 checker.py scan

Test: scan

-------------
Running tests:
-------------

Element Count: 1000000
Exclusive scan outputs are correct!
Correctness passed!
```



```
!cd scan && python3 checker.py scan
Test: scan

--------------
Running tests:
--------------

Element Count: 1000000

Correctness passed!
Student Time: 1.197
Ref Time: 17.357

Element Count: 10000000

Correctness passed!
Student Time: 13.461
Ref Time: 18.108

Element Count: 20000000

Correctness passed!
Student Time: 26.102
Ref Time: 23.238

Element Count: 40000000

Correctness passed!
Student Time: 52.434
Ref Time: 39.807

-------------------------
Scan Score Table:
-------------------------
-------------------------------------------------------------------------
| Element Count   | Ref Time        | Student Time    | Score           |
-------------------------------------------------------------------------
| 1000000         | 17.357          | 1.197           | 1.25            |
| 10000000        | 18.108          | 13.461          | 1.25            |
| 20000000        | 23.238          | 26.102          | 1.25            |
| 40000000        | 39.807          | 52.434          | 0.9489787161002404 |
-------------------------------------------------------------------------
|                                   | Total score:    | 4.698978716100241/5.0 |
-------------------------------------------------------------------------
```



```
!cd scan && ./cudaScan --test find_repeats && ./cudaScan --test find_repeats --arraysize 10000000
---------------------------------------------------------
Found 1 CUDA devices
Device 0: Tesla T4
   SMs:        40
   Global mem: 15095 MB
   CUDA Cap:   7.5
---------------------------------------------------------
Array size: 64
Student GPU time: 0.068 ms
Find_repeats outputs are correct!
```

Task 3:

1. **CUDA Kernel for Circle Rendering (`kernelRenderCircles`):**
 The kernel was modified to properly update pixel colors with atomic operations. Each pixel's RGBA values are updated using atomic addition to ensure thread safety, preventing race conditions during image updates.

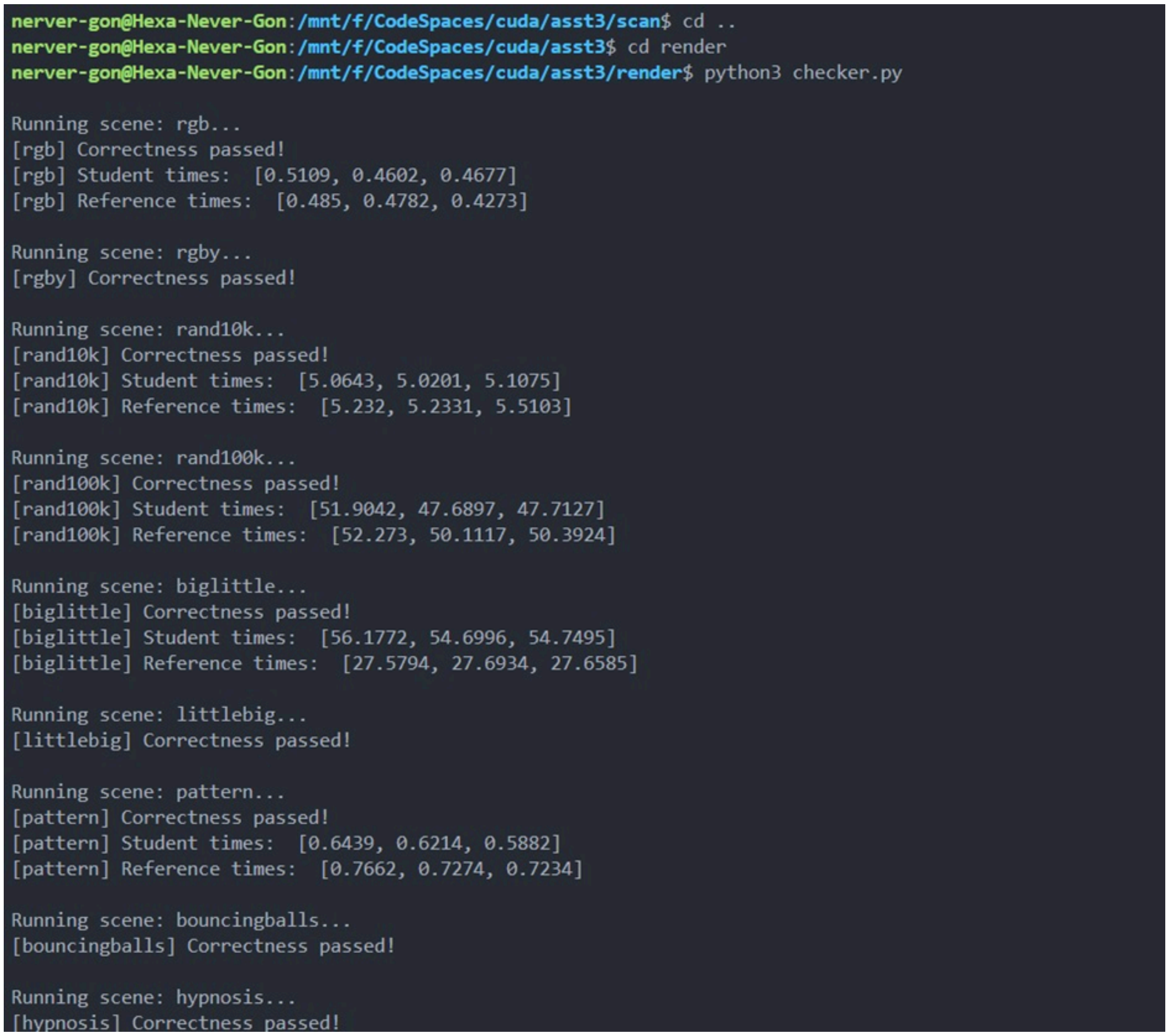
2. **Handling Transparency and Blending:**
 The kernel incorporates blending of transparent circles. The RGBA values are computed for each pixel using the alpha blending formula, ensuring that the rendered

image correctly represents semi-transparent circles.

3. **Ensuring Correct Update Order:**
   The kernel ensures that image updates follow the input order of circles. This is crucial for correct rendering, as overlapping circles should blend in the order they are provided. This step prevents visual artifacts like incorrect overlapping of transparent circles.

4. **Atomicity:**
   Atomic operations are used to guarantee that the read-modify-write cycle on the image pixels is done atomically. This ensures no two threads concurrently modify the same pixel, preserving correctness in multi-threaded execution.

5. **Improved Parallelization:**
   The parallelization of circle processing was optimized by structuring the kernel to handle pixel updates efficiently, distributing the work across GPU threads.



```
nerver-gon@Hexa-Never-Gon:/mnt/f/CodeSpaces/cuda/asst3/scan$ cd ..
nerver-gon@Hexa-Never-Gon:/mnt/f/CodeSpaces/cuda/asst3$ cd render
nerver-gon@Hexa-Never-Gon:/mnt/f/CodeSpaces/cuda/asst3/render$ python3 checker.py

Running scene: rgb...
[rgb] Correctness passed!
[rgb] Student times:  [0.5109, 0.4602, 0.4677]
[rgb] Reference times:  [0.485, 0.4782, 0.4273]

Running scene: rgby...
[rgby] Correctness passed!

Running scene: rand10k...
[rand10k] Correctness passed!
[rand10k] Student times:  [5.0643, 5.0201, 5.1075]
[rand10k] Reference times:  [5.232, 5.2331, 5.5103]

Running scene: rand100k...
[rand100k] Correctness passed!
[rand100k] Student times:  [51.9042, 47.6897, 47.7127]
[rand100k] Reference times:  [52.273, 50.1117, 50.3924]

Running scene: biglittle...
[biglittle] Correctness passed!
[biglittle] Student times:  [56.1772, 54.6996, 54.7495]
[biglittle] Reference times:  [27.5794, 27.6934, 27.6585]

Running scene: littlebig...
[littlebig] Correctness passed!

Running scene: pattern...
[pattern] Correctness passed!
[pattern] Student times:  [0.6439, 0.6214, 0.5882]
[pattern] Reference times:  [0.7662, 0.7274, 0.7234]

Running scene: bouncingballs...
[bouncingballs] Correctness passed!

Running scene: hypnosis...
[hypnosis] Correctness passed!
```