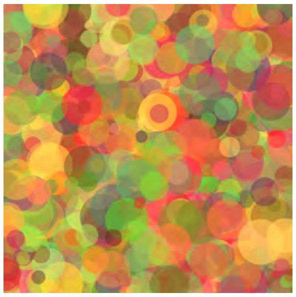
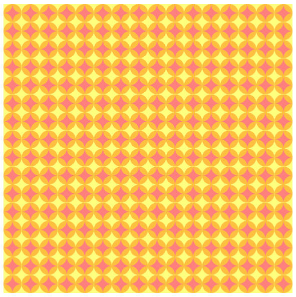
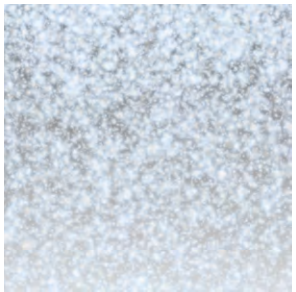
15-418/618, Spring 2026
Assignment 2
A Simple CUDA Renderer

Assigned:	Wed., Jan. 28
Due:	Wed., Feb. 11, 11:59 pm
Last day to handin:	Sat., Feb. 14

Random 10K


Random 100K


Pattern


Snow

Overview

In this assignment you will write a parallel renderer in CUDA that draws colored circles. While this renderer is very simple, parallelizing the renderer will require you to design and implement data structures that can be efficiently constructed and manipulated in parallel. This is a challenging assignment so you are advised to start early. Don't be lulled into thinking that the first two parts are indicative of the time you will require to complete the third part. **Seriously, you are advised to start early.** Good luck!

Before you begin, please take the time to review the course policy on academic integrity at:

`http://www.cs.cmu.edu/˜418/academicintegrity.html`

Environment Setup

You should use the GHC machines containing NVIDIA GeForce RTX 2080 B GPUs. These have host names `ghc`X`.ghc.andrew.cmu.edu`, for X between 1 and 86. The Wikipedia entry for GeForce 20 GPUs provides useful information about this model of GPU. They support CUDA compute capability 7.5.

Please run `nvidia-smi` to confirm that the machine has the correct GPU model.

Using the GHC machines:

1. If you are using the `ssh` command to reach the machine, include the option "`-Y`" in your command line to enable the use of X windows.

2. The NVIDIA CUDA C/C++ Compiler (NVCC) is located at `/usr/local/cuda-11.7/bin/`, which you need to add to your `PATH`. To do this, add the line below to the file below, according to which shell you use. (You can determine which shell you are using with the command "`echo $SHELL`.")

Shell	File	Add
bash	`˜/.bashrc`	`export PATH=/usr/local/cuda-11.7/bin:${PATH}`
csh	`˜/.cshrc`	`setenv PATH /usr/local/cuda-11.7/bin:${PATH}`

3. The CUDA shared library is located at `/usr/local/cuda-11.7/lib64/`. It must be loaded at runtime. Add the line below to your RC file (`˜/.bashrc` or `˜/.cshrc`) according to which shell you use.

```
export LD_LIBRARY_PATH=/usr/local/cuda-11.7/lib64/:${LD_LIBRARY_PATH}
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/usr/local/cuda-11.7/lib64/
```

4. Make sure you source (or reload) your RC file. If using bash, run "`source ˜/.bashrc`." If using csh, run "`source ˜/.cshrc`."

5. Download the Assignment 2 starter code from the course Github using:

```
git clone https://github.com/cmu15418s26/asst2.git
```

Resources

For any C++ questions (like what does the *virtual* keyword mean), the C++ Super-FAQ is a great resource that explains things in a way that's detailed yet easy to understand (unlike a lot of C++ resources), and was co-written by Bjarne Stroustrup, the creator of C++!

The CUDA C programmer's guide is available in PDF or HTML. It is an excellent reference for learning how to program in CUDA.

You can also find a large number of examples in the CUDA SDK `/usr/local/cuda-11.7/samples`. In addition, there are a wealth of CUDA tutorials and SDK examples on the web (just Google!) and on the NVIDIA developer site.

1 CUDA Warm-Up 1: SAXPY (5 pts)

To gain a bit of practice writing CUDA programs your warm-up task is to implement the SAXPY function. This is a function commonly found in matrix math libraries. For input arrays `x` and `y`, output array `dest`, and value `a` (all single-precision floating-point values), the function computes `dest[i] = a∗x[i] +`

y[i] for all array elements i. (The name "SAXPY" stands for "single-precision a times x plus y.") Starter code for this part of the assignment is located in the saxpy directory.

Finish off the implementation of SAXPY in the function saxpyCuda() in saxpy.cu. You will need to allocate device global memory arrays and insert calls to move data between the host and device memories. These issues are covered in Section 3.2.2 of the Programmer's Guide.

As part of your implementation, add timers around the CUDA kernel invocation in saxpyCuda(). After your additions, your program should time two executions:

- The provided starter code contains timers that measure the **entire process** of copying data to the GPU, running the kernel, and copying data back to the CPU.
- Your timers should measure only the time taken to run the kernel. (They should not include the time of CPU to GPU data transfer or transfer of results back to the CPU.)

When adding your timing code, be careful: The CUDA kernel's execution on the GPU is asynchronous with the main application thread running on the CPU. You should place a call to cudaThreadSynchronize() following the kernel call to wait for completion of all CUDA work on the GPU. This call returns only when all prior CUDA work on the GPU has completed. (Without waiting for the GPU to complete, your CPU timers will report that essentially no time elapsed!) Note that in your measurements that include the time to transfer data back to the CPU, a call to cudaThreadSynchronize() *is not* necessary before the final timer (after your call to cudaMemcpy() that returns data to the CPU) because cudaMemcpy() will not return to the calling thread until after the copy is complete.

Question: Compare and explain the difference between the results provided by two sets of timers (the timer you added and the timer that was already in the provided starter code). Are the bandwidth values observed roughly consistent with the reported bandwidths available to the different components of the machine? *Hint*: You should use the web to track down the memory bandwidth of an NVIDIA RTX 2080 GPU, and the maximum transfer speed of the computer's PCIe-x16 bus. It's PCIe 3.0, and a 16 lane bus connecting the CPU with the GPU.

2 CUDA Warm-Up 2: Parallel Prefix-Sum (10 pts)

Now that you're familiar with the basic structure and layout of CUDA programs, as a second exercise you are asked to devise a parallel implementation of the function find_peaks() which, given a list of integers A, returns a list of all indices i for which A[i] is greater than the preceeding and following value. The indices start at 0. Neither the first nor the last element is a peak.

For example, given the array $[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7]$, your program should output the array $[2, 5, 7, 12]$.

All code for this part of the assignment is in the directory scan.

Exclusive Prefix Sum

We want you to implement find_peaks() by first implementing the parallel exclusive prefix-sum operation. This operation is covered in 15-210, where it is referred to by its more general form: the "scan"

```
void exclusive_scan_recursive(int* start, int* end, int* output, int* scratch)
{
    int N = end - start;

    if (N == 0) return;
    else if (N == 1)
    {
        output[0] = 0;
        return;
    }

    // sum pairs in parallel.
    for (int i = 0; i < N/2; i++)
        output[i] = start[2*i] + start[2*i+1];

    // prefix sum on the compacted array.
    exclusive_scan_recursive(output, output + N/2, scratch, scratch + (N/2));

    // finally, update the odd values in parallel.
    for (int i = 0; i < N; i++)
    {
        output[i] = scratch[i/2];
        if (i % 2)
            output[i] += start[i-1];
    }
}
```

Figure 1: Recursive implementation of exclusive prefix sum (for arrays of length $l = 2^k$)

operation.

Exclusive prefix sum takes an array `A` and produces a new array `output` that has, at each index `i`, the sum of all elements up to but not including `A[i]`. For example, given the array $[1, 4, 6, 8, 2]$, the output of exclusive prefix sum $[0, 1, 5, 11, 19]$.

A recursive implementation should be familiar to you from 15-210. As a review (or for those that did not take 15-210), the code in Figure 1 is a C implementation of a work-efficient, parallel version of scan for arrays whose length is a power of 2. Details on prefix-sum (and its more general relative, scan) can be found in Chapter 6 of the 15-210 lecture notes.

While the recursive code expresses our intent well and recursion is supported on modern GPUs, its use can lead to fairly low performance due to function call and stack frame overhead. Instead we can express the algorithm in an iterative manner. The "C-like" code in Figure 2 is an iterative version of `scan`. It operates "in place," meaning that the input data should be copied into the array `data` before the function is called, and the output will be generated in the same array. The code uses `parallel_for` to indicate potentially parallel loops. To understand this code, you may find the visualization in Figure 3 for the case of $\texttt{N} = 16$ helpful.

You are welcome to use this general algorithm to implement a version of parallel prefix sum in CUDA. You

```
void exclusive_scan_iterative(int* data, int* end)
{
    int N = end - data;
    // upsweep phase.
    for (int twod = 1; twod < N; twod*=2)
    {
        int twod1 = twod*2;
        parallel_for (int i = 0; i < N; i += twod1)
            data[i+twod1-1] += data[i+twod-1];
    }
    data[N-1] = 0;

    // downsweep phase.

    for (int twod = N/2; twod >= 1; twod /= 2)
    {
        int twod1 = twod*2;
        parallel_for (int i = 0; i < N; i += twod1)
        {
            int t = data[i+twod-1];
            data[i+twod-1] = data[i+twod1-1];
            // change twod1 below to twod to reverse prefix sum.
            data[i+twod1-1] += t;
        }
    }
}
```

Figure 2: Iterative, in-place implementation of exclusive prefix sum

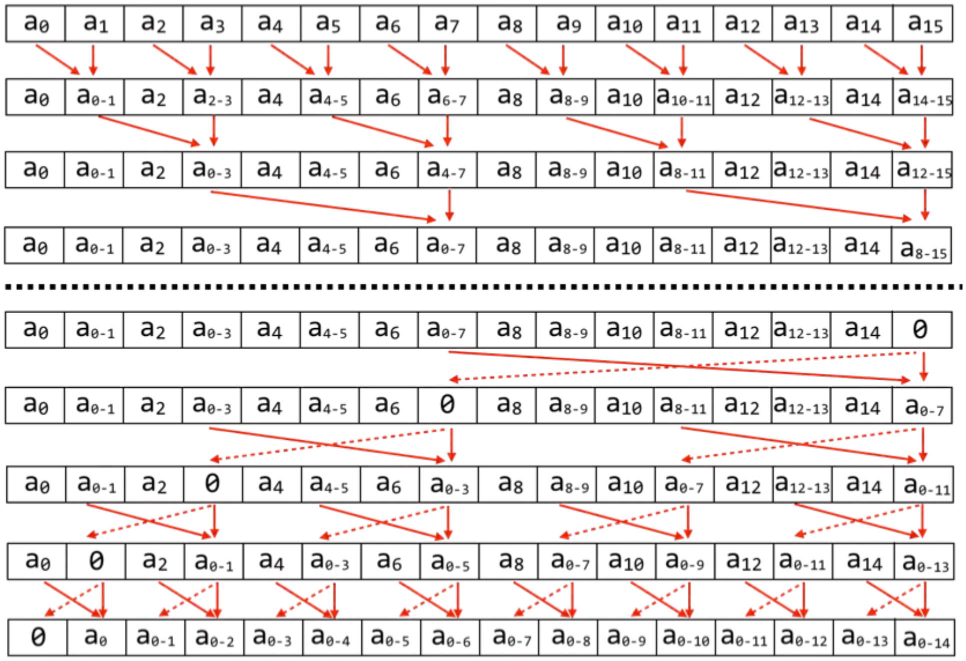

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
a_0 a_{0-1} a_2 a_{2-3} a_4 a_{4-5} a_6 a_{6-7} a_8 a_{8-9} a_{10} a_{10-11} a_{12} a_{12-13} a_{14} a_{14-15}
a_0 a_{0-1} a_2 a_{0-3} a_4 a_{4-5} a_6 a_{4-7} a_8 a_{8-9} a_{10} a_{8-11} a_{12} a_{12-13} a_{14} a_{12-15}
a_0 a_{0-1} a_2 a_{0-3} a_4 a_{4-5} a_6 a_{0-7} a_8 a_{8-9} a_{10} a_{8-11} a_{12} a_{12-13} a_{14} a_{8-15}
a_0 a_{0-1} a_2 a_{0-3} a_4 a_{4-5} a_6 a_{0-7} a_8 a_{8-9} a_{10} a_{8-11} a_{12} a_{12-13} a_{14} 0
a_0 a_{0-1} a_2 a_{0-3} a_4 a_{4-5} a_6 0 a_8 a_{8-9} a_{10} a_{8-11} a_{12} a_{12-13} a_{14} a_{0-7}
a_0 a_{0-1} a_2 0 a_4 a_{4-5} a_6 a_{0-3} a_8 a_{8-9} a_{10} a_{0-7} a_{12} a_{12-13} a_{14} a_{0-11}
a_0 0 a_2 a_{0-1} a_4 a_{0-3} a_6 a_{0-5} a_8 a_{0-7} a_{10} a_{0-9} a_{12} a_{0-11} a_{14} a_{0-13}
0 a_0 a_{0-1} a_{0-2} a_{0-3} a_{0-4} a_{0-5} a_{0-6} a_{0-7} a_{0-8} a_{0-9} a_{0-10} a_{0-11} a_{0-12} a_{0-13} a_{0-14}

Figure 3: Visualization of exclusive prefix sum for 16 elements

must implement the `exclusive_scan()` function in `scan.cu`. Your implementation will consist of both host and device code. The implementation will require multiple kernel launches.

Note: The scan implementation of Figure 2 assumes that the data array's length (`N`) is a power of 2. In the `cudaScan()` function in `scan.cu` (which calls your implementation of `exclusive_scan()`), we solve this problem by rounding the array length to the next power of 2 when allocating the corresponding buffers on the GPU. However, we only copy back `N` elements from the GPU buffer back to the CPU buffer. This fact should simplify your CUDA implementation.

Implementing Find Peaks Using Prefix Sum

Once you have written `exclusive_scan()`, you should implement the function `find_peaks()` in `scan.cu`. This will involve writing more device code, in addition to one or more calls to `exclusive_scan()`. Your code should write the list of indices for the peak values into the provided output array (in device memory), and then return the size of the output list.

When using your `exclusive_scan()` implementation, remember that it operates in place. Also, the array passed to `exclusive_scan()` is assumed to be in `device` memory.

Grading: We will test your code for correctness and performance on random input arrays.

For reference, a scan score table is provided below, showing the performance of a simple CUDA implementation on a Gates machine with a RTX 2080. To check the correctness and performance score of your `scan()` and `find_peaks()` implementations, run the commands:

```
./checker.pl -m scan
./checker.pl -m find_peaks
```

Doing so will produce a reference table like the following:

```
-------------------------
Scan Score Table:
-------------------------
-------------------------------------------------------------------------
| Element Count    | Target Time      | Your Time        | Score           |
-------------------------------------------------------------------------
| 10000            | 0.199            | 0.007 (F)        | 0               |
| 100000           | 0.284            | 0.037 (F)        | 0               |
| 1000000          | 0.465            | 0.077 (F)        | 0               |
| 2000000          | 0.860            | 0.080 (F)        | 0               |
-------------------------------------------------------------------------
|                                     | Total score:     | 0/5             |
-------------------------------------------------------------------------
```

Your score will be based solely on the performance of your code. In order to get full credit, it must perform within 20% of the provided target code.

Test Harness: For grading purposes, the test harness runs on a pseudo-randomly generated array that differs each time it is run. For debugging purposes, you may prefer a more stable environment. Passing the option "`-i test1`" to either `checker.pl` or `cudaScan` will use a fixed, randomly generated input. We encourage you to come up with alternate inputs to your program to help you evaluate it.

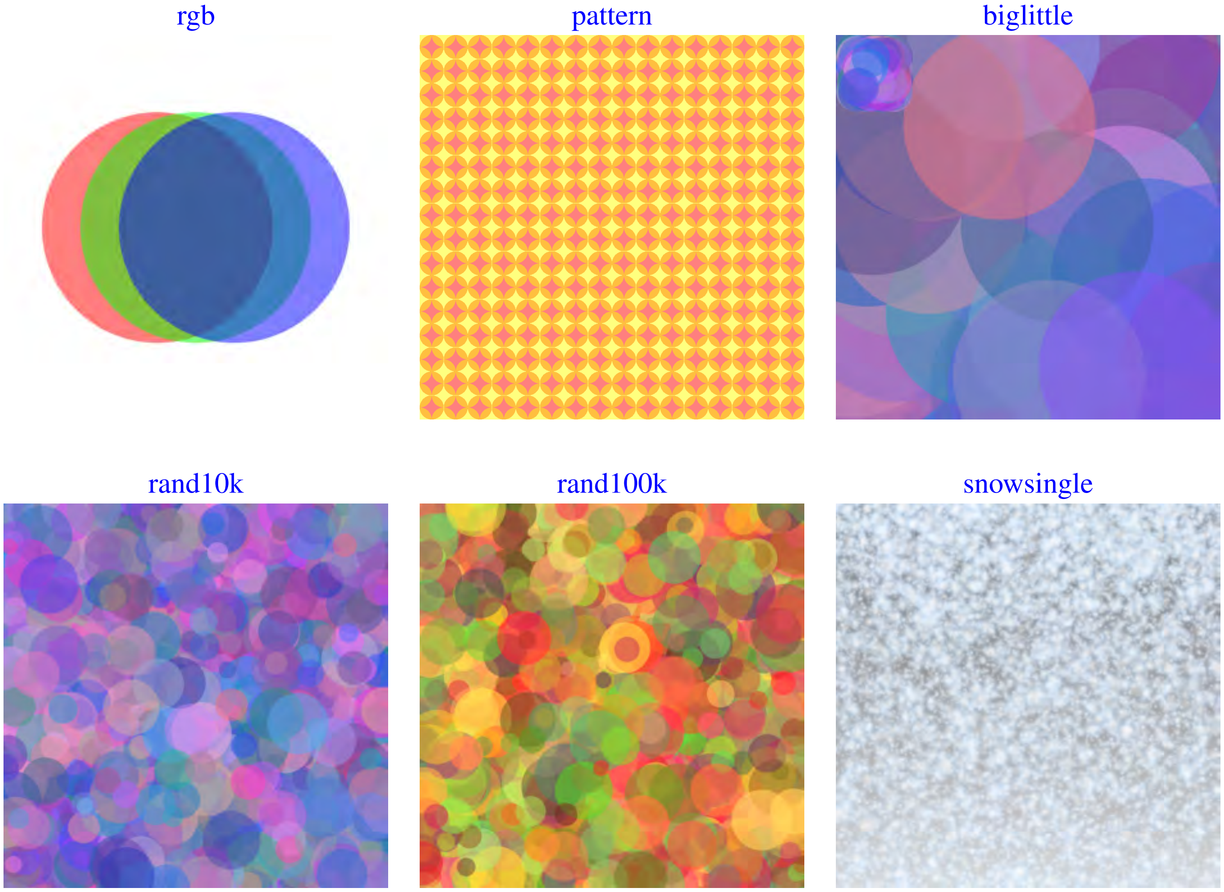
The argument "`-t`" to either `checker.pl` or your compiled program will use the Thrust Library implementation of exclusive scan. We will award up to two points of extra credit for anyone who can create an implementation that is competitive with Thrust.

3 A Simple Circle Renderer (85 pts)

Now for the real show!

The directory `render` of the assignment starter code contains an implementation of a renderer that draws colored circles. Build the code, and run the renderer with the command line "`./render rgb`." You will see an image of three circles appear on screen ('`q`' closes the window). Now run the renderer with the command line "`./render snow`." You should see an animation of falling snow. (You must be either directly using the console of the machine, or include the flag "`-Y`" when connecting to the machine via `ssh`.)

The assignment starter code contains two versions of the renderer: a sequential, single-threaded C++ reference implementation, implemented in `refRenderer.cpp`, and an *incorrect* parallel CUDA implementation in `cudaRenderer.cu`.



rgb

pattern

biglittle

rand10k

rand100k

snowsingle

Figure 4: Sample images generated by renderer

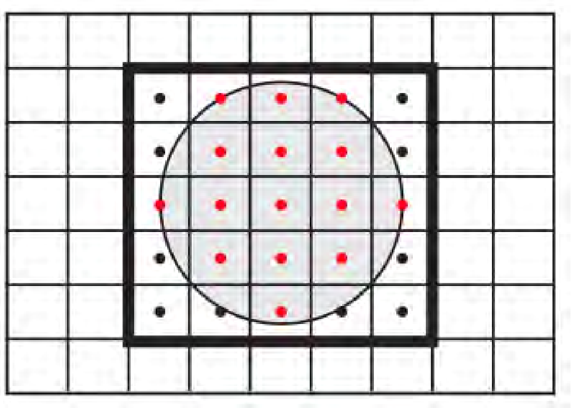


Figure 5: Computing the contribution of a circle to the output image

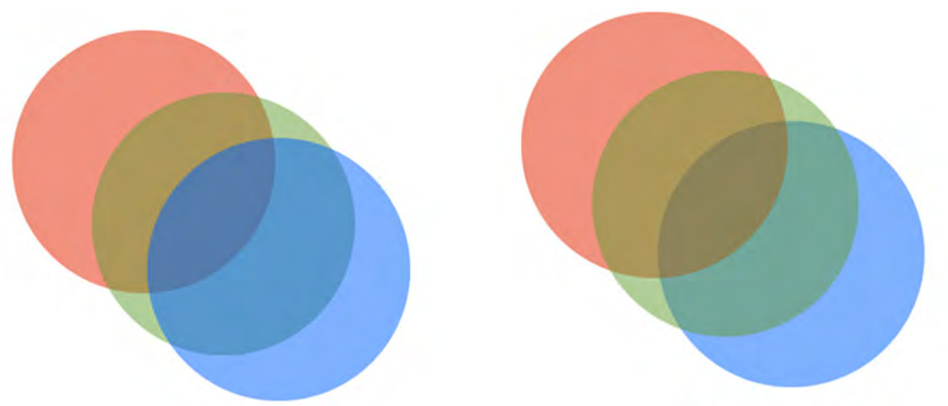
Renderer Overview

We encourage you to familiarize yourself with the structure of the renderer code base by inspecting the reference implementation in `refRenderer.cpp`. The method `setup()` is called prior to rendering the first frame. In your CUDA-accelerated renderer, this method will likely contain all your renderer initialization code (allocating buffers, etc). The `render()` method is called for each frame and is responsible for drawing all circles in the output image. The other main function of the renderer, `advanceAnimation()`, is also invoked once per frame. It updates circle positions and velocities. You will not need to modify `advanceAnimation()` in this assignment.

The renderer accepts an array of circles (3D position, velocity, radius, color) as input. The basic sequential algorithm for rendering each frame is:

```
Clear image
For each circle:
    Update position and velocity
For each circle:
    Compute screen bounding box
    For all pixels in bounding box:
        Compute pixel center point
        If center point is within the circle:
            Compute color of circle at point
            Blend contribution of circle into image for this pixel
```

Figure 5 illustrates the basic algorithm for computing circle-pixel coverage using point-in-circle tests. All pixels within the circle's bounding box are tested for coverage. For each pixel in the bounding box, the pixel is considered to be covered by the circle if its center point (shown with a dot) is contained within the circle.



Correct order:
Red on bottom, blue on top

Incorrect order
Blue on bottom, red on top

Figure 6: Ordering dependency of rendering

Pixel centers that are inside the circle are colored red, while those that are outside the circle are colored black. The circle's contribution to the image will be computed only for covered pixels.

An important detail of the renderer is that it renders **semi-transparent** circles. Therefore, the color of any one pixel is not the color of a single circle, but the result of blending the contributions of all the semi-transparent circles overlapping the pixel (note the "blend contribution" part of the pseudocode above). The renderer represents the color of a circle via a 4-tuple of red (R), green (G), blue (B), and opacity (alpha, written "α") values (RGBA). Alpha value $\alpha = 1.0$ corresponds to a fully opaque circle. Alpha value $\alpha = 0.0$ corresponds to a fully transparent circle. To draw a semi-transparent circle with color (C_r, C_g, C_b, α) on top of a pixel with color P_r, P_g, P_b, the renderer performs the following computation:

$$\begin{aligned} R_r &= \alpha \cdot C_r + (1.0 - \alpha) \cdot P_r \\ R_g &= \alpha \cdot C_g + (1.0 - \alpha) \cdot P_g \\ R_b &= \alpha \cdot C_b + (1.0 - \alpha) \cdot P_b \end{aligned}$$

Notice that composition is not commutative (object X over Y does not look the same as object Y over X), so it's important to render circles in a manner that follows the order they are provided by the application. (You can assume the application provides the circles in depth order.) For example, consider the two images shown in Figure 6, where the circles are ordered as red, green, and blue. The image on the left shows them rendered in the correct order, while the image on the right has them reversed.

CUDA Renderer

After familiarizing yourself with the circle rendering algorithm as implemented in the reference code, now study the CUDA implementation of the renderer provided in `cudaRenderer.cu`. You can run the CUDA implementation of the renderer using the command-line option "`-r cuda`."

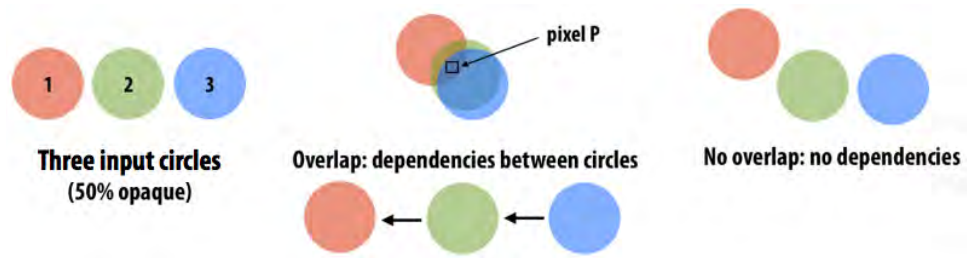

1
2
3
Three input circles
(50% opaque)
pixel P
Overlap: dependencies between circles
No overlap: no dependencies

Figure 7: Illustration of ordering dependencies for rendering

The provided CUDA implementation parallelizes computation across all input circles, assigning one circle to each CUDA thread. While this CUDA implementation is a complete implementation of the mathematics of a circle renderer, it contains several major errors that you will fix in this assignment. Specifically, the current implementation does not ensure that image update is an atomic operation, and it does not preserve the required order of image updates, both of which are described below.

Renderer Requirements

Your parallel CUDA renderer implementation must maintain two invariants that are preserved trivially in the sequential implementation:

1. **Atomicity:** All image update operations must be atomic. The critical region includes reading the four 32-bit floating-point values (the pixel's rgba color), blending the contribution of the current circle with the current image value, and then writing the pixel's color back to memory.
2. **Order:** Your renderer must 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. As discussed above, preserving the ordering requirement allows for a correct rendering of transparent circles. **A key observation is that the definition of order only specifies the order of updates to an individual pixel.** Thus, as shown in Figure 7, there is no ordering requirement between circles that do not contribute to the same pixel. These circles can be processed independently.

Since the provided CUDA implementation does not satisfy either of these requirements, the result of not correctly respecting order or atomicity can be seen by running the CUDA renderer implementation on the different scenes. You may see horizontal streaks through the resulting images, as shown in Figure 8 for the rgb scene. These streaks will change with each frame. If not streaks, you will see cases where the program renders the circles in the wrong order.

What You Need To Do

Your job is to write the fastest, correct CUDA renderer implementation you can. You may take any

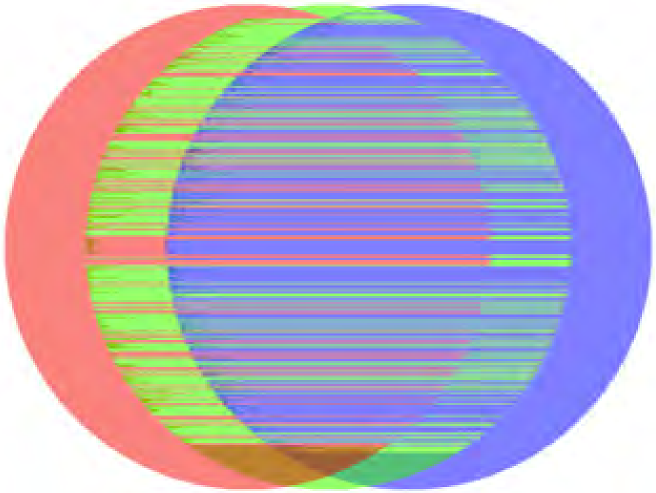


Figure 8: Incorrect rendering by provided CUDA implementation

approach you see fit,[1] but your renderer must adhere to the atomicity and order requirements specified above. A solution that does not meet both requirements will be given no more than 10 points on part 2 of the assignment. We have already given you such a solution!

A good place to start would be to read through `cudaRenderer.cu` and convince yourself that it does not meet the correctness requirement. (Look specifically at the CUDA kernel `kernelRenderCircle`, and the inline function `shadePixel`.) To visually see the effect of violation of above two requirements, compile the program with `make`. Then run:

```
./render -r cuda rand10k
```

Compare this image with the one generated by sequential code by running

```
./render rand10k
```

(This image is shown in the lower-left corner of Figure 4.)

You can get a listing of the options to the `render` program by running

```
./render --help
```

Checker code: To detect correctness of the program, `render` has a convenient "`--check`" option. This option runs the sequential version of the reference CPU renderer along with your CUDA renderer and then compares the resulting images to ensure correctness. The time taken by your CUDA renderer implementation is also printed.

[1] There are two optimizations you may *not* perform: you should render each frame independently, even if one frame contains the same circles as the previous one and you cannot base your optimisations on a specific alpha value.

There are a total of six circle data sets on which you will be graded for performance. However, in order to receive full credit, your code must pass the correctness test for all of our test data sets. There are some hidden tests for preventing you from tailoring the program to specific image size or opacity. The hidden tests have varied image size (w.l.o.g you could assume the size of image is multiple of 64), alpha value, number of circles, etc. To check the correctness and performance score of your code, run “`make check`” (or directly call the program `checker.pl`.) If you run it on the starter code, the program will print the results for the entire test set, plus a table like the following:

```
------------
Score table:
------------
-------------------------------------------------------------------------
| Scene Name      | Target Time     | Your Time       | Score           |
-------------------------------------------------------------------------
| rgb             | 0.1787          | 83.3152 (F)     | 0               |
| rand10k         | 1.6031          | 21.1534 (F)     | 0               |
| rand100k        | 15.1747         | 485.6685 (F)    | 0               |
| pattern         | 0.2175          | 2.4409 (F)      | 0               |
| snowsingle      | 5.2522          | 7.3202 (F)      | 0               |
| biglittle       | 13.9296         | 266.0649 (F)    | 0               |
-------------------------------------------------------------------------
|                                   | Total score:    | 0/72            |
-------------------------------------------------------------------------
```

Note: on some runs, you *may* receive credit for some of these scenes, since the provided renderer’s runtime is non-deterministic. This doesn’t change the fact that the current CUDA renderer is incorrect.

“Target time” is the performance of a good solution on your current machine (in the provided `render_soln` executable.) “Your time” is the performance of your CUDA renderer. Your grade will depend on the performance of your implementation compared to that of the provided implementation (see Grading Guidelines.)

Along with your code, we would like you to hand in a clear, high-level description of how your implementation works as well as a brief description of how you arrived at this solution. Specifically address approaches you tried along the way, and how you went about determining how to optimize your code (For example, what measurements did you perform to guide your optimization efforts?).

Aspects of your work that you should mention in the write-up include:

1. Include both partners’ names and Andrew Id’s at the top of your write-up.
2. Replicate the score table generated for your solution and specify the machine you ran your code on.
3. Describe how you decomposed the problem and how you assigned work to CUDA thread blocks and threads (and maybe even warps.)
4. Describe where synchronization occurs in your solution.
5. What was the main bottleneck in your implementation? How did you identify/measure this bottleneck? Looking back at your implementation, does this lack in performance make sense?

6. What, if any, steps did you take to reduce communication requirements (e.g., synchronization or main memory bandwidth requirements)?
7. Briefly describe how you arrived at your final solution. What other approaches did you try along the way? What was wrong with them?

We highly recommend the use of GPU profiling tools to concretely analyze any existing bottlenecks both when you are completing your write-up **and while you are optimizing your implementation**. Feel free to include metrics and/or screenshots from your profiling results to support your analysis. View Appendix A for more details.

Grading Guidelines

- The write-up for the assignment is worth 13 points.
- Your implementation is worth 72 points. These are equally divided into 12 points per scene as follows:
  - 2 correctness points per scene.
  - 10 performance points per scene (only obtainable if the solution is correct). Your performance will be graded with respect to the target performance of a provided renderer, T_s.
  - No performance points will be given for solutions having time $T > 10T_s$.
  - Full performance points will be given for solutions within 20% of the optimized solution ($T \leq 1.2T_s$.)
  - For other values of T, your performance score on a scale 1 to 9 will be calculated as a linear interpolation based on the value of T_s/T.
  - If your code
    - fails to pass the correctness test for provided test data sets other than above six
    - fails to pass the correctness test for hidden cases
    - has unreasonable performance on hidden cases (e.g. Ts/T halved with small change on image size or alpha)

    10 points per scene will be deducted as a penalty. Up to 20 points in total.
- Up to five points extra credit (instructor discretion) for solutions that achieve significantly greater performance than required. Your write up must clearly explain your approach thoroughly.
- Up to five points extra credit (instructor discretion) for a high-quality parallel CPU-only renderer implementation that achieves good utilization of all cores and SIMD vector units of the cores. Feel free to use any tools at your disposal (e.g., SIMD intrinsics, ISPC, Pthreads.) To receive credit you should analyze the performance of your GPU and CPU-based solutions and discuss the reasons for differences in implementation choices made.

Assignment Tips and Hints

Below are a set of tips and hints compiled from previous years. Note that there are various ways to implement your renderer and not all hints may apply to your approach.

- To facilitate remote development and benchmarking, we have created a "`--bench`" option to the render program. This mode does not open a display, and instead runs the renderer for the specified number of frames. (You'll see that the checking code runs with the command-line option "`--bench 0:4`.")

- When in benchmark mode, the "`--file` *Name*" command-line option sets the base file name for PPM images created at each frame. Created files have names of the form "*Name_xxxx*`.ppm`," where *xxxx* is a 4-digit frame number. No PPM files are created if the `--file` option is not used.

- There are two potential axes of parallelism in this assignment. One axis is parallelism across pixels another is parallelism across circles (provided the ordering requirement is respected for overlapping circles.)

- The prefix-sum operation provided in `exclusiveScan.cu_inl` may be valuable to you on this assignment (you are not required to use it.) It implements exclusive prefix-sum on a **power-of-two-sized** array in shared memory. **The provided code does not work on non-power-of-two inputs and it also requires that the number of threads in the thread block be the size of the array.**

- You are allowed to use the Thrust library in your implementation if you so choose. Thrust is not necessary to achieve the performance of the optimized CUDA reference implementations. See the documentation on prefix-sum operations in Thrust.

- Is there data reuse in the renderer? What can be done to exploit this reuse?

- The circle-intersects-box tests provided to you in `circleBoxTest.cu_inl` are your friend.

- How will you ensure atomicity of image update since there is no CUDA language primitive that performs the logic of the image update operation atomically? Constructing a lock out of global memory atomic operations is one solution, but keep in mind that even if your image update is atomic, the updates must be performed in the required order. **We suggest that you think about ensuring order in your parallel solution first, and only then consider the atomicity problem (if it still exists at all) in your solution.**

- If you are having difficulty debugging your CUDA code, you can use `printf` directly from device code if you use a sufficiently new GPU and CUDA library: see this brief guide on how to print from CUDA.

- If you find yourself with free time, have fun making your own scenes!

Catching CUDA Errors

(Credit: The following was adapted from this Stack Overflow post)

By default, if you access an array out of bounds, allocate too much memory, or otherwise cause an error, CUDA won't normally inform you; instead it will just fail silently and return an error code. You can use the following macro (feel free to modify it) to wrap CUDA calls:

```
#define DEBUG

#ifdef DEBUG
#define cudaCheckError(ans)  cudaAssert((ans), __FILE__, __LINE__);
inline void cudaAssert(cudaError_t code, const char *file, int line, bool abort=true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr, "CUDA Error: %s at %s:%d\n",
        cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}
#else
#define cudaCheckError(ans) ans
#endif
```

Note that you can undefine `DEBUG` to disable error checking once your code is correct for improved performance.

You can then wrap CUDA API calls to process their returned errors as such:

```
cudaCheckError( cudaMalloc(&a, size*sizeof(int)) );
```

Note that you can't wrap kernel launches directly. Instead, their errors will be caught on the next CUDA call you wrap:

```
kernel<<<1,1>>>(a); // suppose kernel causes an error!
cudaCheckError( cudaDeviceSynchronize() ); // error is printed on this line
```

All CUDA API functions, `cudaDeviceSynchronize()`, `cudaMemcpy()`, `cudaMemset()`, and so on can be wrapped.

IMPORTANT: if a CUDA function caused an error previously, but it wasn't caught, that error will show up in the next error check, even if the check wraps a different function. For example:

```
...
line 742: cudaMalloc(&a, -1); // executes, then continues
line 743: cudaCheckError(cudaMemcpy(a,b)); // Prints error for line 743
...
```

Therefore, while debugging, it's recommended that you wrap **all** CUDA API calls (at least in code that you wrote).

Hand-in Instructions

You will submit your code via Autolab and your report via Gradescope. For the code, you will be submitting your entire directory tree. The relevant websites are:

> `https://autolab.andrew.cmu.edu/courses/15418-s26/`
>
> `https://www.gradescope.com/courses/1207156`

1. Your code

   (a) **If you are working with a partner, form a group on Autolab.** Do this before submitting your assignment. One submission per group is sufficient.

   (b) Make sure all of your code is compilable and runnable. We should be able to simply run `make`, then execute your programs in `saxpy`, `scan`, and `render` without manual intervention.

   (c) Remove all nonessential files, especially output images, from your directories.

   (d) Run the command "`make handin.tgz`." This will run "`make clean`" and then create an archive of your entire directory tree.

   (e) Submit the file `handin.tgz` to Autolab.

2. Please upload your report as file `report.pdf` to Gradescope, one submission per team, and select the appropriate pages for each part of the assignment. After submitting, you will be able to add your teammate using the *add group members* button on the top right of your submission.

Our grading scripts will rerun the checker code allowing us to verify your score matches what you submitted in the `report.pdf`. We might also try to run your code on other datasets to further examine its correctness.

Appendix A - Using the Profiler

First, here are two useful video tutorials for Nsight tools:

- `https://www.youtube.com/watch?v=nYSdsJE2zMs`
- `https://www.youtube.com/watch?v=DLQwIdhrL1A` (start at 16:20)

In depth resources about Nsight Compute and Nsight Systems can be found at the following links:

- `https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html`
- `https://developer.nvidia.com/blog/tag/profiling`

Using Nsight Compute

To profile a program, run the command:

```
ncu -o  <reportName> <executable + flags>
```

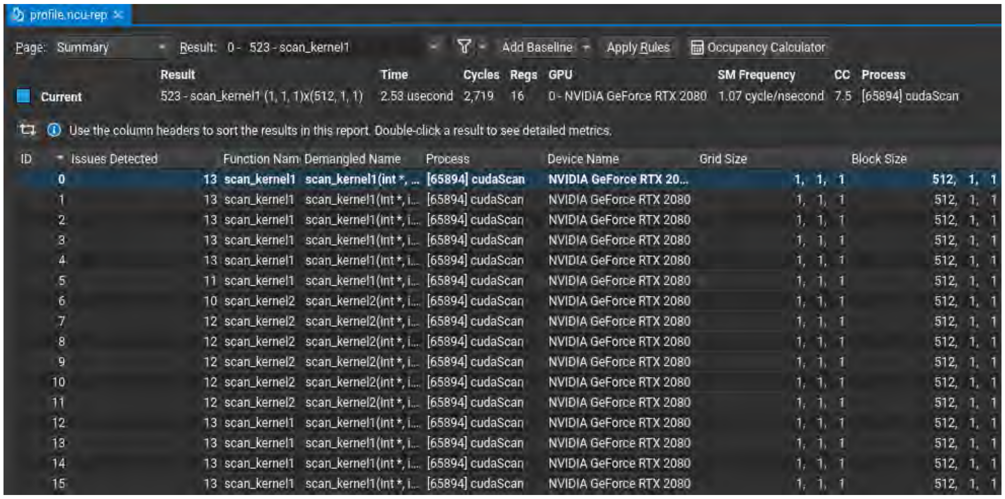
For example, to run this profiler on `cudaScan`, we could run:

```
ncu -o  profile ./cudaScan -m scan -i random
```

where `profile` will be the name of the report file. To view this file, we will use NVIDIA's NCU UI via the following command:

```
ncu-ui <reportName.ncu-rep>
```

Make sure that you have forwarding on in order to view the profiling interface. Upon entering this command, a popup should appear.



Page: Summary   Result: 0 - 523 - scan_kernel1

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	523 - scan_kernel1 (1, 1, 1)x(512, 1, 1)	2.53 usecond	2,719	16	0 - NVIDIA GeForce RTX 2080	1.07 cycle/nsecond	7.5	[65894] cudaScan

Use the column headers to sort the results in this report. Double-click a result to see detailed metrics.

ID	Issues Detected	Function Nam	Demangled Name	Process	Device Name	Grid Size	Block Size
0	13	scan_kernel1	scan_kernel1(int *, ...	[65894] cudaScan	NVIDIA GeForce RTX 20...	1, 1, 1	512, 1, 1
1	13	scan_kernel1	scan_kernel1(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
2	13	scan_kernel1	scan_kernel1(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
3	13	scan_kernel1	scan_kernel1(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
4	13	scan_kernel1	scan_kernel1(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
5	11	scan_kernel1	scan_kernel1(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
6	10	scan_kernel2	scan_kernel2(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
7	12	scan_kernel2	scan_kernel2(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
8	12	scan_kernel2	scan_kernel2(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
9	12	scan_kernel2	scan_kernel2(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
10	12	scan_kernel2	scan_kernel2(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
11	12	scan_kernel2	scan_kernel2(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
12	13	scan_kernel1	scan_kernel1(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
13	13	scan_kernel1	scan_kernel1(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
14	13	scan_kernel1	scan_kernel1(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1
15	13	scan_kernel1	scan_kernel1(int *, i...	[65894] cudaScan	NVIDIA GeForce RTX 2080	1, 1, 1	512, 1, 1

Figure 9: NCU-UI Popup Screen Example

In the top left corner, you can find a drop-down menu labeled 'page.' Here you can choose different parts of the report that the profiler has collected. The *Details* page has the most understandable information, and definitions of terms used can be explained by hovering over the term. The *Source* page can be interesting to look at if lower-level source code is interesting to you. Feel free to explore other pages as well.

To view results without the UI, you can also run:

```
ncu --import <reportName.ncu-rep>
```

Other useful features include `-k` flag for examining specific kernels, `--metrics` flag to print specific values (useful for FLOP/BW analysis), and `--set=full` for more verbose information. Please view the above linked resource for more details.

Important: When running this tool on GHC machines, you may run into the following error:

```
=ERROR= Failed to open/create lock file /tmp/nsight-compute-lock.

Please check that this process has write permissions on this file.
```

This is because the machines do not allow for concurrent use of the profiling tool with multiple users. To avoid this issue, **you should use the provided wrapper script when running Nsight tools** by prefixing your command with `./profile.sh`. For example:

```
./profile.sh ncu -o profile ./cudaScan -m scan -i random
```

If multiple users are profiling at the same time, some performance variability is possible. If you observe unusually high variance in your metrics, try re-profiling or waiting a few minutes.

If you receive a permission error when running the script, make it executable with:

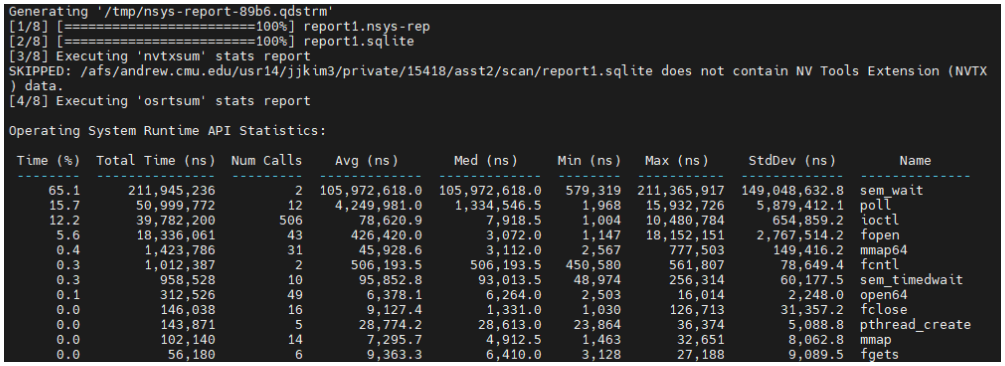
```
chmod +x profile.sh
```

Using Nsight Systems

To view how long specific API calls and GPU activities take, we can use another part of NVIDIA's profiling tools called Nsight systems. To get a view of this data you can use:

```
nsys profile --stats=true <executable>
```

This will give you another breakdown of all the API calls and other stats that occurred during your program's runtime. This command does output two files, but the information in said files is not practical for our purposes. The output of the command should look similar to below:



```
Generating '/tmp/nsys-report-89b6.qdstrm'
[1/8] [========================100%] report1.nsys-rep
[2/8] [========================100%] report1.sqlite
[3/8] Executing 'nvtxsum' stats report
SKIPPED: /afs/andrew.cmu.edu/usr14/jjkim3/private/15418/asst2/scan/report1.sqlite does not contain NV Tools Extension (NVTX
) data.
[4/8] Executing 'osrtsum' stats report

Operating System Runtime API Statistics:

 Time (%)  Total Time (ns)  Num Calls     Avg (ns)       Med (ns)     Min (ns)   Max (ns)     StdDev (ns)        Name
 --------  ---------------  ---------  -------------  -------------  --------  -----------  -------------  --------------
     65.1      211,945,236          2  105,972,618.0  105,972,618.0   579,319  211,365,917  149,048,632.8  sem_wait
     15.7       50,999,772         12    4,249,981.0    1,334,546.5     1,968   15,932,726    5,879,412.1  poll
     12.2       39,782,200        506       78,620.9        7,918.5     1,004   10,480,784      654,859.2  ioctl
      5.6       18,336,061         43      426,420.0        3,072.0     1,147   18,152,151    2,767,514.2  fopen
      0.4        1,423,786         31       45,928.6        3,112.0     2,567      777,503      149,416.2  mmap64
      0.3        1,012,387          2      506,193.5      506,193.5   450,580      561,807       78,649.4  fcntl
      0.3          958,528         10       95,852.8       93,013.5    48,974      256,314       60,177.5  sem_timedwait
      0.1          312,526         49        6,378.1        6,264.0     2,503       16,014        2,248.0  open64
      0.0          146,038         16        9,127.4        1,331.0     1,030      126,713       31,357.2  fclose
      0.0          143,871          5       28,774.2       28,613.0    23,864       36,374        5,088.8  pthread_create
      0.0          102,140         14        7,295.7        4,912.5     1,463       32,651        8,062.8  mmap
      0.0           56,180          6        9,363.3        6,410.0     3,128       27,188        9,089.5  fgets
```

Figure 10: NSYS Terminal Output Example

Important: Please conduct these profilers on terminating code (eg. use the `-b` (bench) option for the render assignment). Also note for the `./render` executable, make sure to include the `-r cuda` option such that the code is run through a CUDA runtime.

Nsight Troubleshooting

Verify that your environment has been properly set up as specified in the handout. If problems persist, run

```
    cat /proc/driver/nvidia/params | grep RmProfilingAdminOnly
```

and verify that this is set to 0. If not, please make a Piazza post to let us know, then try a different machine. For questions about `profile.sh`, please ask on Piazza.