# EE488 Lab 1. CUDA Programming

**KAIST EE**
**Course Instructor: Minsoo Rhu**

Part 3 of this lab was graciously provided by Kayvon Fatahalian of CMU; originally developed for Kayvon's "Parallel Computer Architecture and Programming" course. Do not share the material from this lab outside of class.

**(Important Note!)** All evaluations should be evaluated and reported in your write-up using the **NVIDIA GTX 1070** as configured in the PCs at Haedong lounge. This is to compare the performance of your CUDA implementation against your fellow students, so MAKE SURE YOU COLLECT YOUR FINAL NUMBERS USING THE GPUS AT **HAEDONG LOUNGE**! Failure to abide by this rule will **penalize** your Lab 1 score by **20%**.

## Overview

Random 10K Random 100K Pattern Snow

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. Seriously, you are advised to start early. Good luck!

## Environment Setup

This lab assignment requires a GPU with CUDA compute capability so as announced during the first lecture, please use the PCs available at Haedong lounge if you don't already have access to a GPU yourself. Once logged on to a Linux server, you can check the current status of the GPUs available in your computer using the following command.

```
$> nvidia-smi
```

CUDA related tools are usually located in the following path:

```
/usr/local/cuda
```

Make sure the CUDA_PATH variable is set properly in the Makefile of any code you are trying to compile. In general, the PCs at Haedong lounge had the CUDA tools installed under: CUDA_PATH ?= "/usr/local/cuda/".

If, for some reason, the environment variables are not properly setup (e.g., $CUDA_PATH is not set to /usr/local/cuda) and you cannot use CUDA, (1) download the 'ee488.source' file from KLMS, (2) copy it into your home directory, and (3) use the following command.

```
$> source ~/ee488.source
```

## Reference Materials for CUDA

The CUDA C programmer's guide is an excellent reference for learning how to program in CUDA.

The latest CUDA version is 9.0 and most (if not all) of the computers at Haedong lounge are installed with CUDA 9.0. Some machines do have 8.0 installed in them and in general, this should not be a problem in going through this lab assignment. But in any case, if you have trouble compiling your source code, try using another PC or ask the TAs.
You can also learn by looking at the sample code that NVIDIA provides.

```
/usr/local/cuda/samples
```

There are also many tutorials and courses online. For example, this course on Udacity.

## Building and Running Starter Code

Download the tarball file from the LMS system and copy it under your target directory, for instance:

```
/home/YOUR_ID/lab_1/…
```

For instance, your directory should look something like `/home/minsoorhu/lab_1/1_saxpy` … If you would like to build the starter code, just go into each directory and enter 'make'.

## Part 1: CUDA Warm-Up 1: SAXPY

To gain a bit of practice writing CUDA programs your warm-up task is to implement the SAXPY function (`result = scale*X+Y`, where `X`, `Y`, and `result` are vectors of N elements and `scale` is a scalar value). Starter code for this part of the assignment is located in the `/1_saxpy` directory of the tarball.

Please finish off the implementation of SAXPY in the function saxpyCuda in saxpy.cu. You will need to allocate device global memory arrays and copy the contents of the host input arrays `X`, `Y`, and `result` into CUDA device memory prior to performing the computation. After the CUDA computation is complete, the `result` must be copied back into host memory. Please see the definition of cudaMemcpy function 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

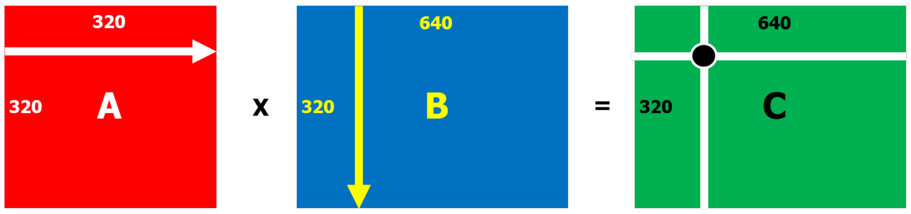
- The provided starter code contains timers that measure **the entire process** of copying data to the GPU, running the kernel, and copying data back to the CPU.
- Your timers should measure only the time taken to run the kernel. (They should not include the time of CPU to GPU data transfer or transfer 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. Therefore, you will want to place a call to `cudaThreadSynchronize` following the kernel call to wait for completion of all CUDA work on the GPU. This call to `cudaThreadSynchronize` returns 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 cudaMemcopy that returns data to the CPU) because cudaMemcpy will not return to the calling thread until after the copy is complete.

**Question 1.** What performance do you observe compared to the sequential CPU-based implementation of SAXPY? 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 the NVIDIA GPU you've used for your experiment, and the maximum transfer speed of the computer's PCIe bus. It's **PCIe 3.0**, and a 16 lane bus connecting the CPU with the GPU.)

## Part 2: CUDA Warm-Up 2: Matrix-Multiplication

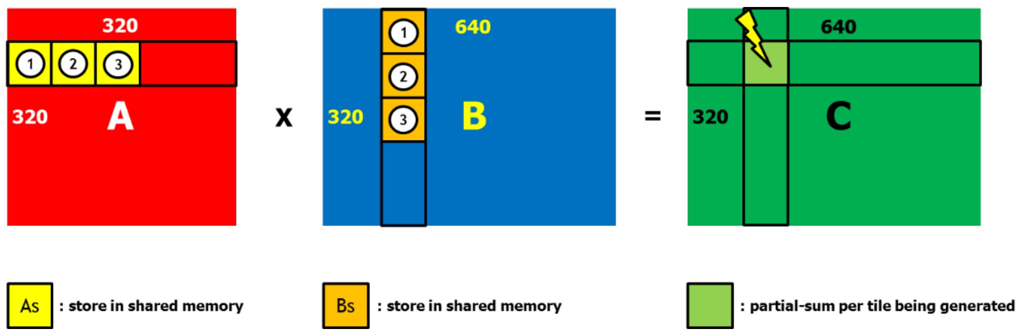
Now that you're familiar with the basic structure and layout of CUDA programs, as a second exercise, you are asked to implement the matrix-multiplication algorithm (i.e., A*B=C) as discussed during our class. For the purpose of this assignment (for both Part (a) and Part (b) below), we assume the size of matrix A, B, and C is (320,320), (640,320) and (640, 320), respectively as shown below. Each thread-block is configured as dim3 threads(BLOCK_SIZE, BLOCK_SIZE, 1), which means that each thread-block contains (BLOCK_SIZE*BLOCK_SIZE) number of CUDA threads. The value of BLOCK_SIZE is defined as a macro inside matrixMul.cu file.



### (a) "Naïve" implementation of matrix-multiplication

We want you to first implement the matrix-multiplication based on a brute-force approach, which means that each CUDA thread brings in the entire elements within a given row in A (the left-to-right arrow in matrix A above) as well as the entire elements within a given column in B (the top-to-bottom arrow in matrix B above), deriving the final output scalar value inside matrix C (the indices of which is (row_id, column_id)). You need not have to worry about optimizations such as blocking (aka tiling) the operands inside on-chip storage – the purpose of this assignment is to familiarize yourself with the key algorithmic nature of matrix multiplication how you should go about mapping it into CUDA.

### (b) "Better" implementation of matrix-multiplication using shared-memory



As an optimization on top of the naïve version of matrix multiplication, we will leverage shared memory (aka scratchpad memory) as means to increase (compute:memory) ratio and better utilize off-chip memory bandwidth. As shown in the figure above, rather than having each CUDA thread individually derive a single scalar output in matrix C, we will have a group of CUDA threads (in our case, all BLOCK_SIZE*BLOCK_SIZE threads within a thread-block) cooperatively bring in the BLOCK_SIZE number of rows (the rectangular strip) in matrix A and also the BLOCK_SIZE number of columns (the rectangular strip) in matrix B into shared memory. These two strips are used to derive the [BLOCK_SIZE x BLOCK_SIZE] sized output tile in matrix C as shown in the above figure. You are asked to properly allocate the shared memory space (i.e., As[] and Bs[] above) and sequence the data fetch (which is, bringing in A[] and B[] from off-chip memory to shared memory As[] and Bs[]) and the corresponding computation steps in a race-condition free manner (remember we talked about `__synchthreads()`?).

**Question 2.** What performance do you observe for your naïve and shared-memory based implementation of matrix-multiplication? How far off are these numbers when compared against the reference_solution (the `./matrixMul_reference` executable file under `/2_matrixMul` directory)? If the performance gap between your version and the reference solution is wide, why do you think that is (HINT: think about the trade-off between thread-level-parallelism and shared-memory data reuse behavior in our matrix-multiplication algorithm)? How far off are your performance numbers when compared against the theoretical maximum single-precision throughput available with the GPU you've used for this lab?

## Part 3: A Simple Circle Renderer

Now for the real show!

The directory `/3_renderer` of the assignment starter code contains an implementation of renderer that draws colored circles. Build the code, and run the render with the following 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.

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 incudaRenderer.cu.

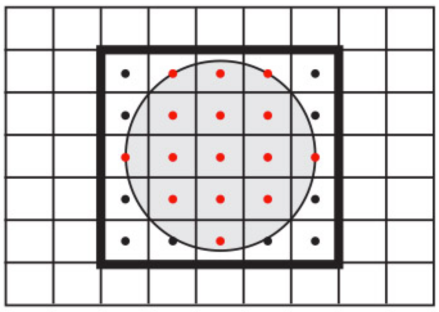
### Renderer Overview

We encourage you to familiarize yourself with the structure of the renderer codebase 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). render is called each frame and is responsible for drawing all circles into 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
```

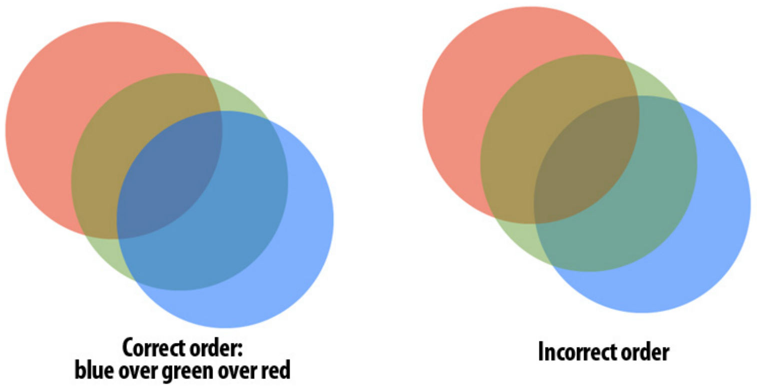
The figure below illustrates the basic algorithm for computing circle-pixel coverage using point-in-circle tests. Notice that a circle contributes color to an output pixel only if the pixel's center lies within the circle.



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) values (RGBA). Alpha = 1 corresponds to a fully opaque circle. Alpha = 0 corresponds to a fully transparent circle. To draw a semi-transparent circle with color (C_r, C_g, C_b, C_alpha) on top of a pixel with color (P_r, P_g, P_b), the renderer uses the following math:

```
   result_r = C_alpha * C_r + (1.0 - C_alpha) * P_r
   result_g = C_alpha * C_g + (1.0 - C_alpha) * P_g
   result_b = C_alpha * C_b + (1.0 - C_alpha) * P_b
```

Notice that composition is not commutative (object X over Y does not look the same as object Y over X), so it's important that the render draw 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 below. In the image on the left, the circles are drawn in the correct order. In the image on the right, the circles are drawn out of order.



**CUDA Renderer**

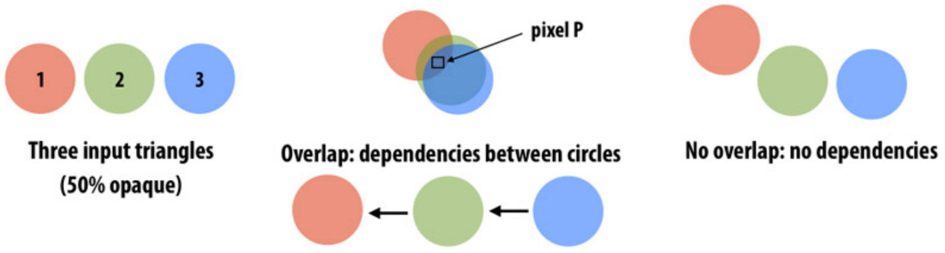
After familiarizing yourself with the circle rendering algorithm as implemented in the reference code, now study the CUDA implementation of the renderer provided in cudaRenderer.cu. You can run the CUDA implementation of the renderer using the --renderer cuda program option.

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 image update is an atomic operation and it does not preserve the required order of image updates (the ordering requirement will be described below).

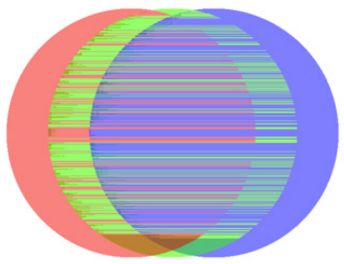
**Renderer Requirements**

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 correct rendering of transparent circles. **A key observation is that the definition of order only specifies the order of updates to the same pixel.** Thus, as shown in Figure 4 below, there are no ordering requirements between circles that do not contribute to the same pixel. These circles can be processed independently.


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

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 rgb and circles scenes. You will see horizontal streaks through the resulting images, as shown in Figure 5 below. These streaks will change with each frame.



**What You Need To Do**

**Your job is to write the fastest, correct CUDA renderer implementation you can.** You may take any approach you see fit, 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 12 points on part 3 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. To visually see the effect of violation of above two requirements, compile the program with make. Then run ./render -r cuda rgb which should display the three circles image. Compare this image with the image generated by sequential code by running ./render rgb.

Following are some of the options to ./render:

```
-b  --bench START:END    Benchmark mode, do not create display. Time frames from START to END
-c  --check              Runs sequential and cuda versions and checks correctness of cuda code
-f  --file  FILENAME     Dump frames in benchmark mode (FILENAME_xxxx.ppm)
-r  --renderer WHICH     Select renderer: WHICH=ref or cuda
-s  --size  INT          Make rendered image x pixels
-?  --help               Prints information about switches mentioned here.
```

**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.

We provide a total of five circle datasets you will be graded on. However, in order to receive full credit, your code must pass all of our correctness-tests. To check the correctness and performance score of your code, run **./checker.pl** in the /render directory. If you run it on the starter code, the program will print a table like the following, along with the results of our entire test set:

```
------------
Score table:
------------
--------------------------------------------------------------------------
| Scene Name      | Fast Time (Tf)    | Your Time (T)     | Score          |
--------------------------------------------------------------------------
| rgb             | 0.2286            | 134.3623 (F)      | 0              |
| rand10k         | 2.8718            | 41.7973 (F)       | 0              |
| rand100k        | 28.6707           | 1101.6551 (F)     | 0              |
| pattern         | 0.3399            | 2.2731 (F)        | 0              |
| snowsingle      | 14.9097           | 11.0902 (F)       | 0              |
| biglittle       | 22.2598           | 559.4087 (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.

"Fast time" is the performance of a good solution on your current machine (in the provided render_ref executable). "Your time" is the performance of your current CUDA renderer solution. Your grade will depend on the performance of your implementation compared to these reference implementations (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. Replicate the score table generated for your solution and specify which machine you ran your code on.
2. Describe how you decomposed the problem and how you assigned work to CUDA thread blocks and threads (and maybe even warps).
3. Describe where synchronization occurs in your solution.
4. What, if any, steps did you take to reduce communication requirements (e.g., synchronization or main memory bandwidth requirements)?
5. Briefly describe how you arrived at your final solution. What other approaches did you try along the way. What was wrong with them?

**Grading Guidelines**

- Maximum points achievable through this assignment: 100 points
  - Saxpy: 5 points
    - Hand-in write-up for Question 1 (2 points)
    - Pass (3 points) or fail (0 points)
  - Matrix-multiplication: 25 points
    - Hand-in write-up for Question 2 (5 points)
    - Functional correctness (10 points)
      - Naïve (3 points) and shmem (7 points), Pass (max) or fail (0 points)
    - Performance points (10 points): Maximum 5 points for each naïve and shmem
      - Full performance points will be given for solutions within 20% of the optimized, reference solution ( $T < 1.20 * T_{ref}$ )
      - If the correctness test failed, 0 points for the performance points
  - Renderer: 70 points
    - Hand-in write-up: 10 points
    - Your implementation is worth 60 points. These are equally divided into 10 points per scene (6 scenes total) as follows:
      - 2 correctness points per scene.
      - 8 performance points per scene (only obtainable if the solution is correct). Your performance will be graded with respect to the performance of a provided benchmark reference renderer, $T_{ref}$:
      - No performance points will be given for solutions having time (T) an order of 10 times the magnitude of $T_{ref}$.
      - Full performance points will be given for solutions within 20% of the optimized solution ( $T < 1.20 * T_{ref}$ )
      - For other values of T (for $1.20 T_{ref} <= T < 10 * T_{ref}$), your performance score on a scale 1 to 10 will be calculated as: $10 * T_{ref}/T$.
    - 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.

**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 --benchmark option to the render program. This mode does not open a display, and instead runs the renderer for the specified number of frames.
- When in benchmark mode, the --file option sets the base file name for PPM images created at each frame. Created files are basename xxxx.ppm. 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).
- Is there data reuse in the renderer? What can be done to exploit this reuse?
- 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**.

## Catching CUDA Errors

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

```
...
line 742: cudaMalloc(&a, -1); // executes, then continues
line 743: cudaCheckError(cudaMemcpy(a,b)); // prints "CUDA Error:
out of memory at cudaRenderer.cu:743"
...
```

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

(Credit: adapted from **this Stack Overflow post**)