Parallel Graphics Renderer using CUDA

Joel Jacob and Yadnesh Samant

ABSTRACT

To utilize the CUDA toolkit to parallelize the task of rendering geometric shapes. We employ the GPU on the NVidia Jetson Tx1 Embedded platform to perform the rendering process. We compare the performance between rendering on the CPU and the GPU.

INTRODUCTION

Rendering is a process of generating an image from 2D models or by utilizing algorithms. The result of such a model can be called as rendered image. This process also involves adding shades, color to a 2-D or 3-D in order to create life-like images on a screen. Real time rendering is often used for 3-D video games and creating highly interactive scenes. This is a growing field. With a large number of gaming enthusiasts, life like visualizations become important.

Rendering is a highly compute intensive in graphical processing, with most of the computation being done in floating point units. It is very important to parallelize this work as well as make it as fast as possible, so that there is no slowdown or lag. The CPU falls short in this aspect, the GPU performs well in this scenario, as most of the rendering process can be parallelized with a high comparative speedup. These aspects caught out attention and hence our decision to work on CUDA, which is NVidia's GPU based development language.

CUDA

CUDA stands for compute unified device architecture. It is a parallel computing platform and application programming by NVidia. This platform is designed to work with programming language such as C, C++. We have used CUDA C++ and compiled it using NVCC i.e. NVidia's CUDA compiler. We have run all the rendering application on the Jetson TX1 embedded platform and used the CUDA 8.0 Toolkit.

Basic Flow of CUDA processing:

- Copy data from host (CPU) memory to GPU memory.
- Instruct the GPU by instantiating and calling the Kernels.
- Let GPU execute the kernels on each CUDA cores.

- Copy the results from GPU memory into the CPU (host) memory.

NVCC

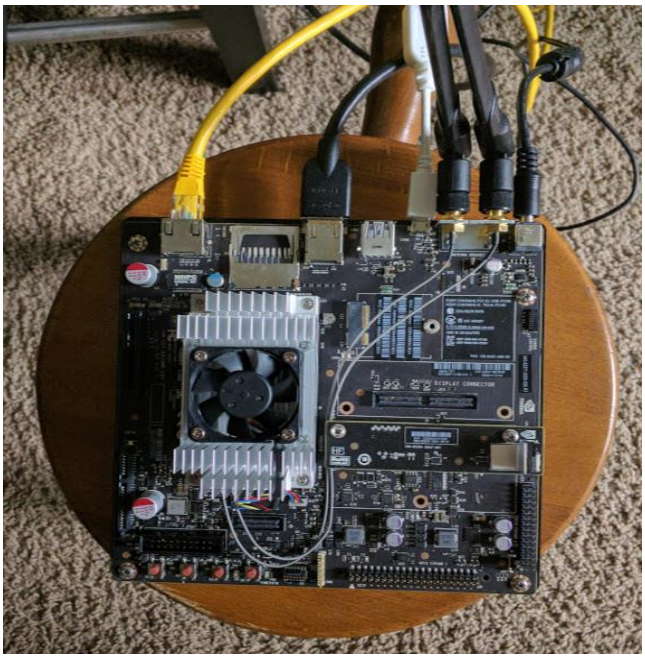
NVCC stands for NVidia's CUDA compiler. CUDA runs on both CPU and GPU. NVCC separates the code into host code and device code. The host code runs on the CPU while the device code is run on the GPU. As we used CUDA C++, the host code is compiled using gcc compiler. The device code is compiled by NVCC.

Nsight IDE

NVidia's Nsight is an integrated development environment for compiling and debugging both CPU and GPU code. It also offers other features like optimizing performance, understanding how the algorithm can be optimized for better performance and analyzing of bottlenecks and see the system behavior.

We have used Nsight eclipse on linux system, which helped us in creating and debugging the renderer. We also ran the profiler to understand the system timing and performance of GPU compared to CPU. The results are discussed in later part of the paper.

JETSON TX1



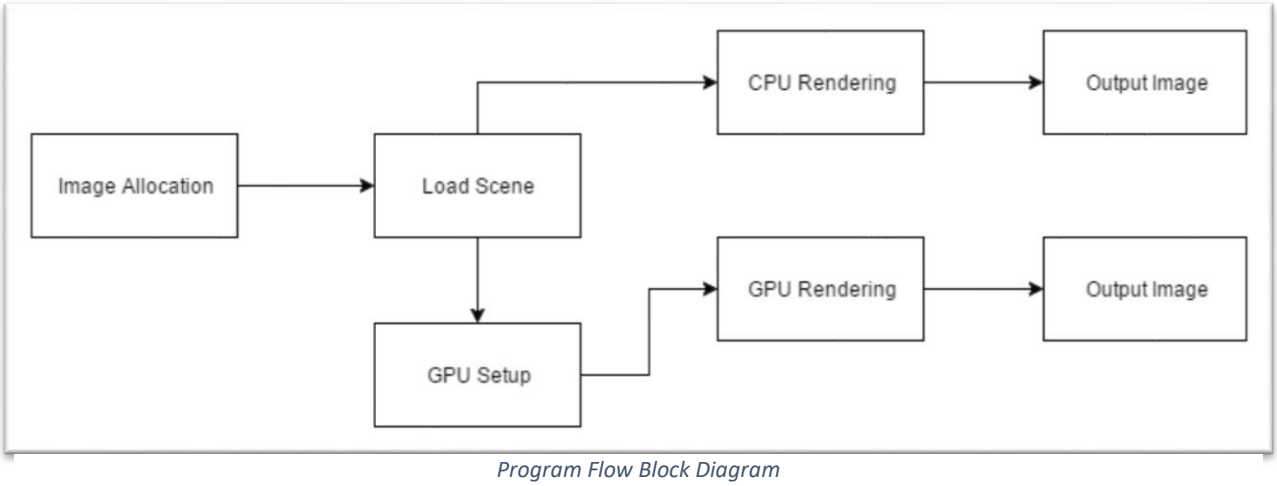
Jetson TX1 Developer Kit

This kit is a full-featured development platform for AI computing and CUDA programming applications. It comes pre-flashed with a Linux environment, includes support for many common APIs, and is supported by NVIDIA's complete development tool chain.

It uses the NVidia Maxwell architecture and has 256 CUDA cores. The CPU is a Quad core ARM A57 chip.

Jetson Jetpack

Jetson jetpack is a complete SDK used for developing applications on NVidia's embedded platform. It



Program Flow Block Diagram

installs developer tools on both the host PC and the Jetson TX1 including the libraries and APIs.

DESIGN IMPLEMENTATION

PROGRAM FLOW:

The algorithm follows following steps:

- We generate circles.
- We randomly position, assign a color to each circle. The depth is the number of that circle (for simplicity).
- We also specify a circle radius.
- We then render the circle in memory using both the CPU and GPU.
- We then dump these rendered values into image files of fixed dimensions.

BLOCK DIAGRAM:

- A frame of specified resolution is allocated to memory.
- The resources for the scene to be loaded are also stored in memory.
- For the GPU, there's a setup process that copies over contents of the scene and allocated frame into GPU device memory and we also set some constants in the GPU registers.
- We render the image using both the CPU and GPU and dump those pixels onto their respective image frame that sits in memory.

UNDERSTANDING THE RENDERER FLOW:

The renderer accepts the array of circles with 3 co-ordinates namely x, y and depth representing the z-

axis. We also provide the color and radius for each circle.

Initially, the image file is cleared to generate a clean starting point for every simulation. We then create a bounded box representing the height and width as per resolution of the image required. For each pixel in the Box, we see if it lies within circle parameters. If so, we compute a color for that pixel.

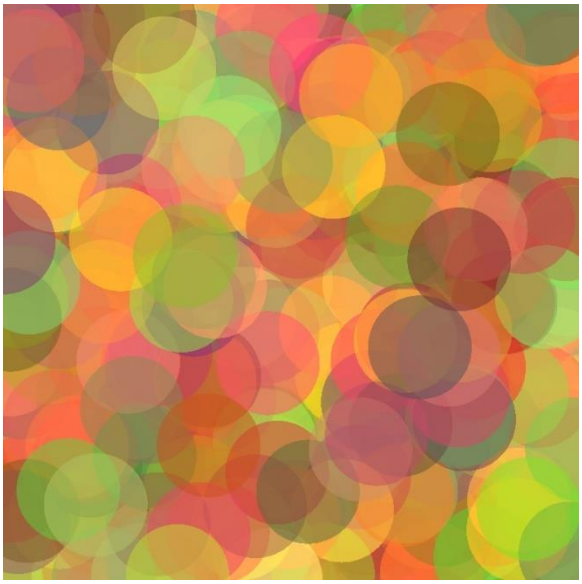
For computation of the color we considered transparency and overlap of different circles into account. The new color will depend on a gradient factor called alpha.

```
newColor.x = alpha * rgb.x + oneMinusAlpha * existingColor.x;
newColor.y = alpha * rgb.y + oneMinusAlpha * existingColor.y;
newColor.z = alpha * rgb.z + oneMinusAlpha * existingColor.z;
```

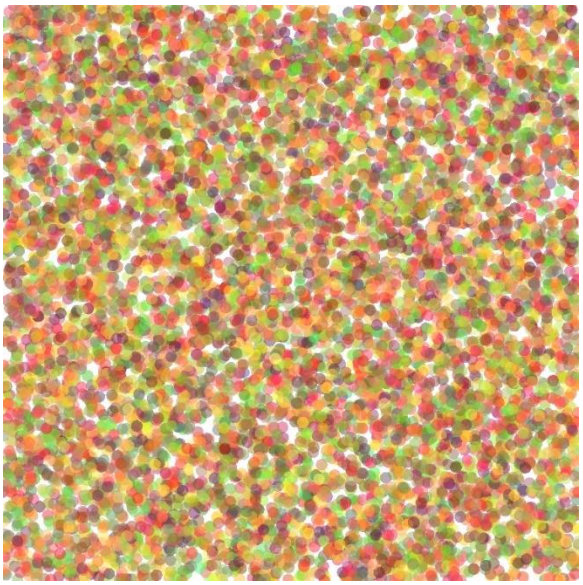
RENDERED IMAGES:

The images were rendered by varying different parameters, namely, the radius of circle, the number of circles rendered and the resolution of the screen. This was done to understand how our implementation works for different benchmarks which define graphics. The following images show various scaling parameters and the output of the rendered image. We have also verified that the output rendered by CPU and GPU is same in visualization, but we improve by 5x in timing when it comes to running on GPU.

Random radii



50K circles



Radius = 0.01

Radius = 0.15

RESULTS:

Radius (10Kcircles)	CPU	CUDA	Speedup
0.01	0.3727	0.366	1.018306
0.05	0.4388	0.201	2.1830846
0.09	10.7	2.1	5.0952381
0.15	27.52	5.12	5.375

As noticed from the table above, at lower radius values the speedup isn't significant as the CUDA kernel has some overhead due to copy functions between the host CPU to GPU memory.

AS the radius increases there is a stark difference in the speedup obtained when the images are rendered on both the CPU and GPU. There is a speedup of around 5 throughout after the initial hiccup.

No. of Circles (Radius = 0.09)	CPU	CUDA	Speedup
1000	1.21	0.4065	1.018306
10000	10.7	2.1	5.0952381
25000	26.48	6.6	4.0121212
50000	52.9	13.15	4.0228136

As the radius remains constant and the circle number varies, we see the speedup is low initially and then we

obtain a speedup of 5 for 10K circles, which looks to be a sweet spot for our application.

As the number of circles increases, the GPU overhead reduces the speedup somewhat.

Resolution (10Kcircles & Rad = 0.09)	CPU	CUDA	Speedup
768*768	10.7	2.1	5.0952381
1024*1024	18.76	4.138	4.4848195
2048*2048	74.81	14.87	5.0309348

In this case, we only vary the image resolution. The boundary normalization condition used in the image causes the speedup to remain almost the same throughout.

CHALLENGES

- New hardware platform pains!
- Things don't "just" work out of the box.
- Remote debugging can be painful.
- Lots of documentation to wade through.
- Learning to debug smart. Kernels and threads can overwhelm quick!

CONCLUSION

We were able to successfully render circles in parallel using both the CPU and GPU by utilizing the CUDA Toolkit.

We were also able to compare the performance between rendering the image both on the CPU and GPU and obtained a speedup of 5.

As the scale was normalized, our rendered circles match in ratio with the resolution of the screen. This was also verified with the speedup obtained for different resolutions that were considered, which remains constant with varying width and height of screen.

REFERENCES

- http://en.wikipedia.org
- http://15418.courses.cs.cmu.edu/spring2017/article/4
- https://developer.nvidia.com/embedded/develop/tools
- http://www.nvidia.com/object/nsight.html
- https://developer.nvidia.com/embedded/develop/tools