Insper

# Relatório do Projeto de Ray Tracing implementado em GPU

**Nome: Hugo Mendes**
**Professor: Luciano Soares**
**Disciplina: Supercomputação**

**7º Semestre do curso de Engenharia de Computação**

# Proposta do projeto

A proposta deste projeto é dar sequência à uma série de aprendizados e técnicas/ferramentas utilizados na área de supercomputação. Sendo um desses temas, o uso de placas gráficas para a realização de tarefas pesadas, em sua maior parte contas.

Sendo uma sequência do projeto de ray tracing implementado na primeira metade do semestre, esse projeto tem como intuito a mesma tarefa de renderizar uma cena de computação gráfica com o algoritmo de ray tracing mas agora paralelizado na gpu. Como adendo, vale a comparação de performance entre ambos algoritmos.

# Material

Os testes implementados para esse relatório foram feitos em uma máquina virtual da AWS, sendo o sistema operacional da máquina um Ubuntu versão 16.03 LTS tendo como placa de vídeo uma v100 da nvidia. O tipo da instância é p2.xlarge.

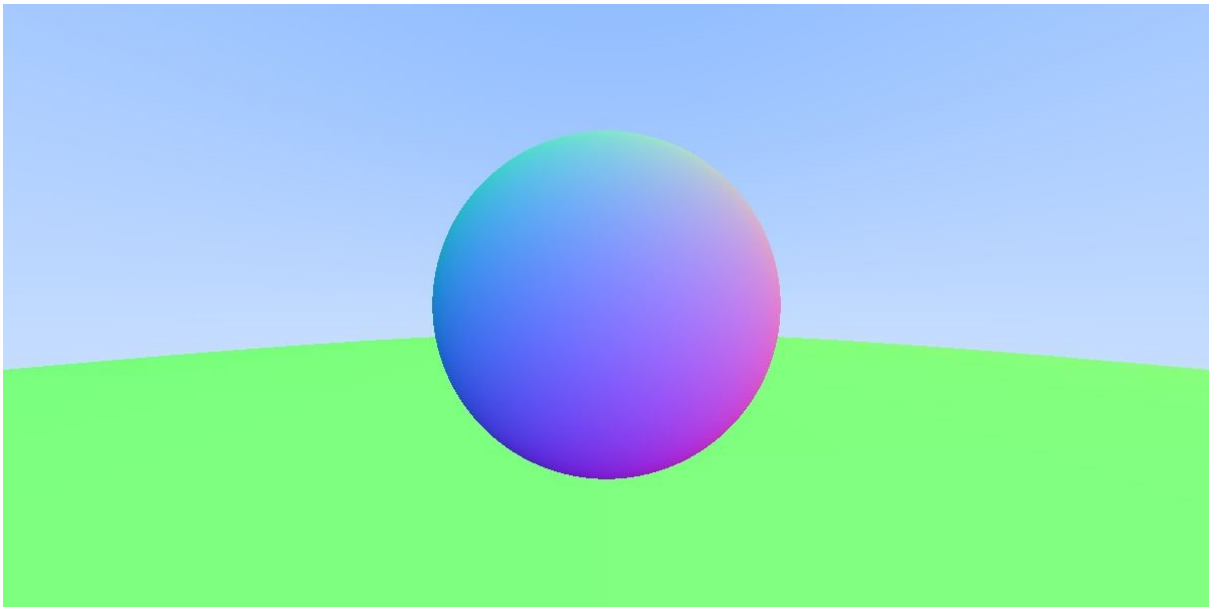
# Metodologia

A máquina disponibilizada para testes estava com seus recursos sendo compartilhado por outros estudantes, o que pode gerar dados inconsistentes e tempos de execução maiores do que o esperado. A metodologia para geração dos resultados, foi para uma mesma resolução de imagem (x por y) medir via clock dentro do próprio programa o tempo de execução do programa, sendo o tempo a primeira e a última coisa do programa a serem calculados. Para destrinchar o tempo observado em mais detalhes, como por exemplo o tempo de execução em cada função do algoritmo foi usado o excelente profiler da nvida, o nvprof.

# Resultados

Segue abaixo a imagem gerada pelo algoritmo e cujo tempos de processamento destrinchados em sub-níveis serão baseados. Vale a pena lembrar

que este é um algoritmo de ray-tracing simplificado e não implementa antialiasing.



Duas esferas geradas com o algoritmo de raytracing.

Os tipos de dados coletados para comparação foram:

- Tempo de processamento absoluto e percentual na gpu da função render
- Tempo de processamento absoluto e percentual na gpu da função create_world
- Tempo de processamento absoluto e percentual de chamadas de api do cuda:
    - MallocManaged
    - DeviceReset
    - DeviceSynchronize
- Tempo total de processamento do programa

Variantes:

- Resolução (em pixels)
- Processamento na GPU / CPU

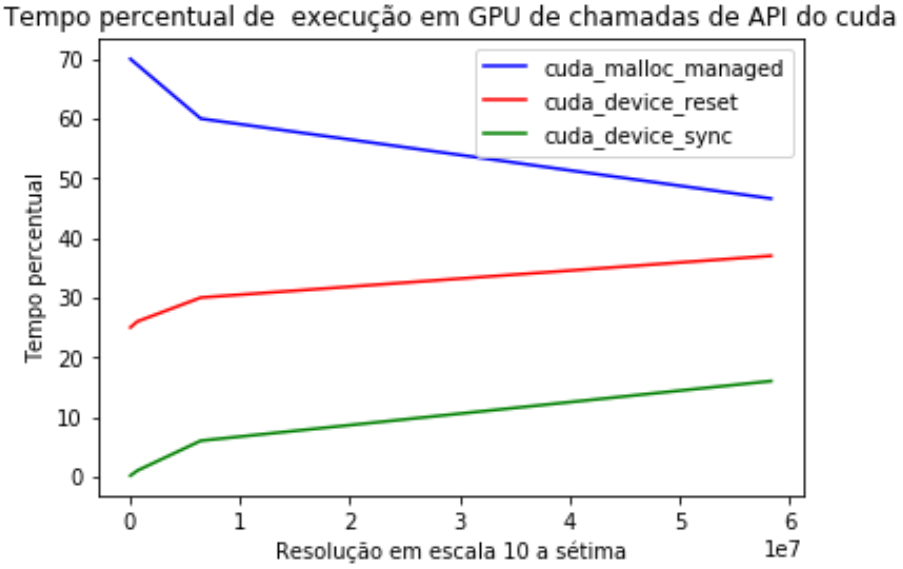
A tabela abaixo mostra os resultados de performance observados nas funções do algoritmo:

| Resolução /Funções | render (s) | render (%) | create_world (s) | create_world (%) |
|---|---|---|---|---|
| 400x200 | 0.000213 | 77.75 | 0.000061 | 22.25 |
| 1200x600 | 0.00183 | 96.8 | 0.000060 | 3.20 |
| 3600x1800 | 0.016 | 99.63 | 0.000061 | 0.37 |
| 10800x5400 | 0.129 | 99.95 | 0.000061 | 0.05 |

**Tabela de performance de funções do algoritmo.**

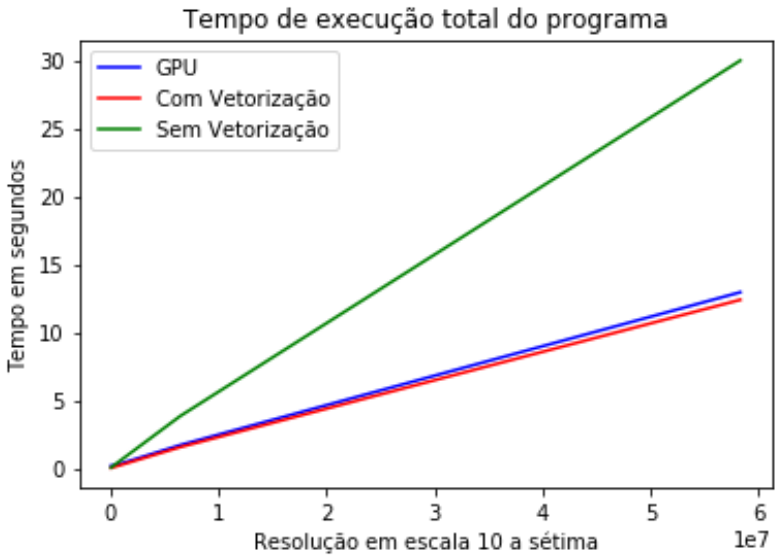
Percebe-se que o gargalo de performance, ou, o código a ser otimizado e tratado com atenção está dentro da função render, uma vez que quando escalando esses números a função create_world tende a ficar mais próxima de 0. O que já era esperado, uma vez que todas as operações aritméticas mais intensas estão dentro da função render.

Analisando agora os métodos de chamada de api do cuda, estes foram os resultados:




As 3 chamadas de api são as que mais tomam tempo de processamento, e percebe-se que a medida que o tamanho da imagem aumenta mais tempo demora para resetar o dispositivo gráfico e sincronizar as threads que nele rodam.

Chegando à camada mais externa para análise, o tempo de processamento total de um programa, os resultados surpreendem. Segue um gráfico que explicita essa relação:




Como o esperado, o código totalmente puro rodando na cpu foi o que pior performou dentro dessa análise, e surpreendentemente, para os tamanhos de imagem utilizados, um código vetorizado (que já permite uma certa característica paralela) rodando em cpu performou tão bem quanto um código rodando na gpu. A hipótese para explicar esse comportamento é a simplicidade da cena, a gpu deveria se destacar em tempo quando se renderizando mais esferas com diferentes tipos de material.

# Reprodutibilidade

Todos os códigos utilizados para a geração desses dados podem ser encontrados em:

https://github.com/hugosoftdev/RaytracingGPUCUDAMASTERLOL

Código este que foi baseado em dois projetos muito conhecidos de ray tracing e que também podem ser encontrados em:

RaytracingCpu: https://github.com/petershirley/raytracinginoneweekend

RaytracingGpu: https://github.com/rogerallen/raytracinginoneweekendincuda/tree/ch06_antialiasing_cuda

# Conclusão

Após a realização destes testes ficou evidente a necessidade de uma decisão sábia e sob estudos de quando realmente deve-se desenvolver uma solução baseada em código para rodar em gpu. Deve-se analisar principalmente, qual a natureza do algoritmo e o tamanho do dado que será computado.

Novamente  pode-se perceber o porque a velocidade de escrita e leitura na memória tem tanto impacto. Custa muito caro (em processamento e também em dinheiro) a transferência de uma massa grande de dados via barramento para componentes distantes. A GPU lida bem com operações simples porém numerosas, mas também tem-se todo um trabalho do desenvolver de converter um código de cpu para cuda, o que geralmente não é fácil. Em resumo, são vários trade-offs que devem ser estudados antes da adoção rodar códigos em gpu para um projeto.

# Referências Bibliográficas


https://github.com/rogerallen/raytracinginoneweekendincuda/tree/ch06_antialiasing_cuda

https://github.com/petershirley/raytracinginoneweekend

https://www.nvidia.com.br/object/prbr_05282018.html

https://devblogs.nvidia.com/accelerated-ray-tracing-cuda/