Relatório Projeto 2

Supercomputação

Antonio Lacombe Sigrist

São Paulo, 6 de junho de 2019

Sumário

1. Ray Tracing Algorithm

O Ray Tracing é um algoritmo descrito pela primeira vez em 1969 por Arthur Appel. A ideia desse algoritmo é renderizar imagens 3D a partir de "raios de luz" traçados que saiam do objeto e vão até os olhos de quem olha. Esse método permite acompanhar a trajetória de raios de luz e entender o caminho e encontros que ele teve com diferentes objetos até chegar no nosso olho, o que compromete a forma com que vemos ele. Esse método é reconhecido pela sua capacidade de simular efeitos de luz, sombra e contrastes de forma mais precisa e possibilitar um cenário mais imersivo e próximo da realidade.

Para cada pixel da imagem, é lançado um raio de luz que sai em direção ao foco fora da tela representado pelo olho do observador representado pela função lookat(). A partir daí, a imagem começa a ser varrida pixel por pixel da esquerda para direita, de cima para baixo analisando o que acontece com cada raio de luz que de lá sai indo para direção determinada. O algoritmo leva em consideração o tipo do material que a luz sai e incide no caminho, como metal e vidro por exemplo.

Essa técnica é muito usada para criar ambientes em games e em uma tela de 1920 x 1080 pixels, mais de 2 milhões de pixels precisam ser atualizados 30 vezes por segundo (maioria dos games exigem o dobro), enquanto o jogo está em execução. Isso é um exemplo da importância de otimização desse algoritmo, para que ele seja capaz de gerar a imagem o mais rápido suficiente.

A proposta desse projeto é usar algumas técnicas de superprogramação para deixá-lo mais eficiente no que se diz a tempo de geração de imagem. Analisaremos também o impacto que a otimização traz na necessidade de memória.

Para entender mais a fundo o código:

https://www.youtube.com/watch?v=bN8AV_x4BXI

https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/how-does-it-work

https://www.techtudo.com.br/noticias/2018/04/o-que-e-ray-tracing-veja-como-funciona-a-tecnologia-para-placas-de-video.ghtml

1.1 Objetivo do projeto

Tendo em vista a importância de conseguir um tempo de processamento rápido, capaz de processar milhões de pixels por segundo (processamento de imagem em tempo real), vamos utilizar nesse segundo projeto um hardware conhecido na superprogração pela sua hiper capacidade de processamento conhecido por GPU. A ideia é que, realizando um mix de processamento entre a CPU e GPU, consiga-se obter um tempo de processamento desejado para fins como *games, etc.*

2. GPU contra CPU

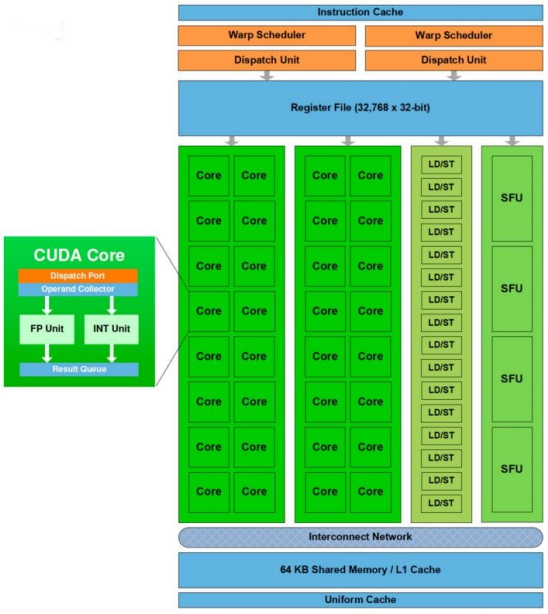
A CPU é composta de apenas alguns núcleos com muita memória cache que pode lidar com alguns threads de software ao mesmo tempo. Já uma GPU é composta de centenas de núcleos que podem manipular milhares de threads simultaneamente. A capacidade de uma GPU com mais de 100 núcleos para processar milhares de threads pode otimizar alguns softwares em mais de 100x apenas com uma CPU. Além disso, a GPU atinge essa enorme desempenho com uma boa eficiência em termos de energia e de custo comparado a uma CPU.

No PC de hoje, a GPU agora pode assumir muitas tarefas multimídia, como acelerar o vídeo Adobe Flash, transcodificar (traduzir) vídeo entre diferentes formatos, reconhecimento de imagem, correspondência de padrões de vírus e outros. Cada vez mais, os problemas realmente difíceis de resolver são aqueles que têm uma natureza inerentemente paralela, como processamento de vídeo, análise de imagem e processamento de sinal.

A combinação de uma CPU com uma GPU pode oferecer o melhor valor de desempenho, preço e potência do sistema.

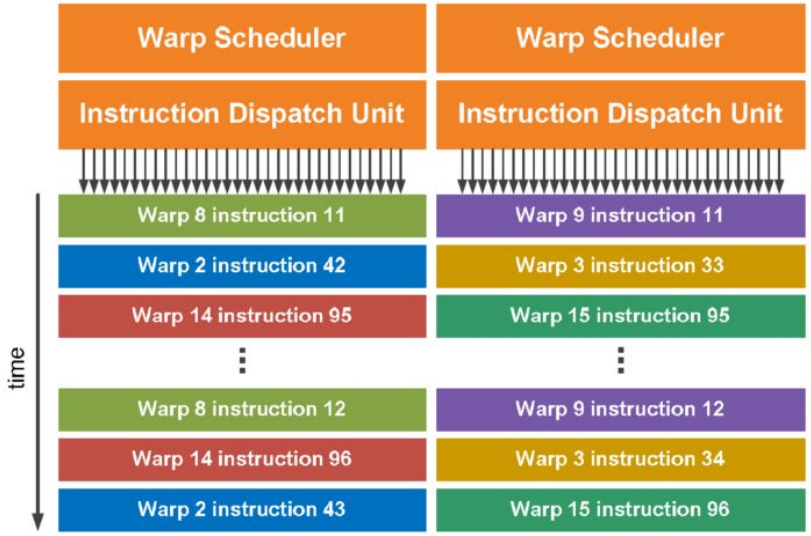
2.1 Arquitetura de uma GPU

A imagem a seguir mostra a arquitetura Fermi. Esta GPU tem 16 multiprocessadores de streaming (SM), que contém 32 núcleos cuda cada um. Cada cuda é uma unidade de execução para números inteiros e flutuantes.



Fermi Streaming Multiprocessor (SM)

A próxima imagem nos permite observar que cada SM tem 32 núcleos cuda, 2 Warp Scheduler e unidade de despacho, um monte de registradores, 64 KB de memória compartilhada configurável e cache L1. Os núcleos Cuda são a unidade de execução que possui um processador de computação float e um processador de computação inteiro. O SM programa threads em um grupo de 32 threads chamados warps. O Warp Schedulers significa que dois warps podem ser emitidos ao mesmo tempo.



3. Codando em GPU

3.1 Otimizações Realizadas

A ideia de transpor o código de c++ para Cuda se baseia na ideia de quebrar o processamento dos raios traçados em diversas threads. Dessa maneira, cada thread trataria o que acontece no percurso de cada raio, e bastaria à CPU buscar o dado já tratado para retornar a imagem.

Para isso, devemos garantir que todas as threads estejam "*settadas*" com os mesmo estados para garantir que o cenário sendo tratado por cada thread é o mesmo e eles não vão gerar imagens diferentes por threads. Para isso, deveremos nos certificar que todas threads terão a mesma seed e universo, que será configurado via kernels.

Para realizar as adaptações do código em C++ para o código em Cuda, foi utilizado um tutorial fornecido pelo blog da Nvidia que pode ser encontrado nas referências. O código da Nvidia oferece uma maneira para evitar/descobrir/tratar erros que possam ser gerados nas comunicações de API entre host e device, mas optei por não implementá-las visto que podem gerar perda de velocidade de processamento. Como estamos fazendo o processamento de uma única cena e não de um game em tempo real, as chances de erro de API de comunicação são muito menores e puderam ser desprezadas. Não obtive nenhum erro durante as compilações.

Para realizar as mudanças para GPU, devemos inicialmente entender quais são as funções que serão rodadas na CPU e na GPU. Para isso, é necessário trabalhar com a ideia de __host__ e __device__, identificando os métodos que são chamados pela CPU (*host*) e pela GPU (*device*). Tendo isso em vista, em frente a todos métodos é necessário adicionar '__host__ __device__' para garantir que ambas máquinas poderam utilizar os métodos.

```
public:
    __host__ __device__ vec3() {}
    __host__ __device__ vec3(float e0, float e1, float e2) { e[0] = e0; e[1] = e1; e
    __host__ __device__ inline float x() const { return e[0]; }
    __host__ __device__ inline float y() const { return e[1]; }
    __host__ __device__ inline float z() const { return e[2]; }
    __host__ __device__ inline float r() const { return e[0]; }
    __host__ __device__ inline float g() const { return e[1]; }
    __host__ __device__ inline float b() const { return e[2]; }

    __host__ __device__ inline const vec3& operator+() const { return *this; }
    __host__ __device__ inline vec3 operator-() const { return vec3(-e[0], -e[1], -e
    __host__ __device__ inline float operator[](int i) const { return e[i]; }
    __host__ __device__ inline float& operator[](int i) { return e[i]; };

    __host__ __device__ inline vec3& operator+=(const vec3 &v2);
    __host__ __device__ inline vec3& operator-=(const vec3 &v2);
    __host__ __device__ inline vec3& operator*=(const vec3 &v2);
    __host__ __device__ inline vec3& operator/=(const vec3 &v2);
    __host__ __device__ inline vec3& operator*=(const float t);
    __host__ __device__ inline vec3& operator/=(const float t);

    __host__ __device__ inline float length() const { return sqrt(e[0]*e[0] + e[1]*e
    __host__ __device__ inline float squared_length() const { return e[0]*e[0] + e[1
    __host__ __device__ inline void make_unit_vector();


    float e[3];
};


inline std::istream& operator>>(std::istream &is, vec3 &t) {
    is >> t.e[0] >> t.e[1] >> t.e[2];
    return is;
```

Outro ponto que se deve mudar é a forma em que é traçado raios. Sabemos que os raios são traçados de maneira aleatória, e como precisamos criar esses estados aleatórios dentro das threads do Cuda, precisaremos gerar utilizando uma biblioteca própria dele. A sugerida pela roteiro foi utilizar uma biblioteca chamada de *curand.* Inicialmente, criamos ela na primeira thread zero do bloco zero e vamos usar esse mesmo estado nas outras threads para garantir a consistência da imagem.

```
__global__ void cria_cena(curandState *rand_state,hitable **d_list,
hitable **d_world, camera **d_camera, int nx, int ny) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        curand_init(seed, 0, 0, rand_state); //cria uma seed na thread 0
que vai garantir mesma seed para todas as threads

        curandState local_rand_state = *rand_state;
        d_list[0] = new sphere(vec3(0,-1000.0,-1), 1000, new
```

```
lambertian(vec3(0.5, 0.5, 0.5)));
        int i = 1;
        for(int a = -11; a < 11; a++) {
            for(int b = -11; b < 11; b++) {
                float choose_mat = RND;
                vec3 center(a+RND,0.2,b+RND);
                if(choose_mat < 0.8f) {
                    d_list[i++] = new sphere(center, 0.2, new
lambertian(vec3(curand_uniform(&local_rand_state)*curand_uniform(&local_
rand_state), RND*RND, RND*RND)));
                }
                else if(choose_mat < 0.95f) {
                    d_list[i++] = new sphere(center, 0.2, new
metal(vec3(0.5f*(1.0f+RND), 0.5f*(1.0f+RND), 0.5f*(1.0f+RND)),
0.5f*RND));
                }
                else {
                    d_list[i++] = new sphere(center, 0.2, new
dielectric(1.5));
                }
            }
        }

        d_list[i++] = new sphere(vec3(0, 1,0),  1.0, new
dielectric(1.5));
        d_list[i++] = new sphere(vec3(-4, 1, 0), 1.0, new
lambertian(vec3(0.4, 0.2, 0.1)));
        d_list[i++] = new sphere(vec3(4, 1, 0),  1.0, new
metal(vec3(0.7, 0.6, 0.5), 0.0));

        *rand_state = local_rand_state;
        *d_world  = new hitable_list(d_list, 22*22+1+3);

        vec3 lookfrom(25,12,13);
        vec3 lookat(0,0,0);
        float dist_to_focus = 10.0; (lookfrom-lookat).length();
        float aperture = 0.1;
        *d_camera   = new camera(lookfrom,
                                 lookat,
                                 vec3(0,1,0),
                                 30.0,
                                 float(nx)/float(ny),
                                 aperture,
                                 dist_to_focus);
```

```
    }
}
```

Nota-se que nesse mesmo kernel estamos criando a cena inicial que será repassada para todas as threads. Isso foi feito diferentemente do tutorial que realizava suas chamadas diferentes de kernels realizando testes de erros que diminuiam a performance do processamento. Para evitar ficar realizando comunicações entre host e device, desejamos replicar o cenário para cada uma das threads para que todas threads possam rodar independentemente com todos dados necessários em si. Enquanto para C++ utilizávamos a função de número pseudo aleatório 'drand48()', em Cuda devemos substituí-la por curand_uniform(&local_rand_state). Também, por mais que alguns Cudas mais recentes já processem *double*, muitos não usam essa precisão. Por isso é recomendado nos dados de pontos flutuantes de processamento em Cuda utilizar floats (ex 1.0f).

Uma vez que esse estado e cenário já está criados, devemos repassar esse estado randômico gerado pela seed para todas as threads de processamento via a render_init:

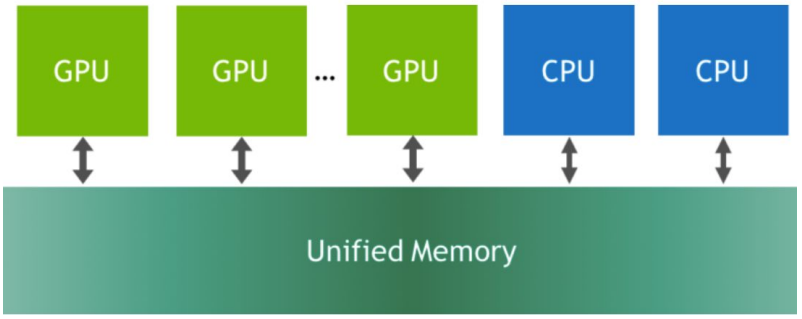
```
__global__ void render_init(int max_x, int max_y, curandState
*rand_state) {
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   int j = threadIdx.y + blockIdx.y * blockDim.y;
   if((i >= max_x) || (j >= max_y)) return;
   int pixel_index = j*max_x + i;

   curand_init(seed, pixel_index, 0, &rand_state[pixel_index]);
}
```

Com todas threads devidamente inicializadas, podemos aplicar a função render sobre cada uma das threads. Para isso, devemos novamente criar um kernel da função render, criando uma cópia local em cada thread para ser processado. Nota-se que esse kernel recebe e passa para cada thread o ponteiro que aponta para o estado inicial gerado pela seed e para o cenário criado pela create world para que cada thread independente possa realizar seu processamento sem depender de mais nenhum outro dado.

```
__global__ void render(vec3 *fb, int max_x, int max_y, int ns,
```

```
camera **cam, hitable **world, curandState *rand_state) {
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   int j = threadIdx.y + blockIdx.y * blockDim.y;
   if((i >= max_x) || (j >= max_y)) return;
   int pixel_index = j*max_x + i;
   curandState local_rand_state = rand_state[pixel_index];
   vec3 col(0,0,0);
   for(int s=0; s < ns; s++) {
      float u = float(i + curand_uniform(&local_rand_state)) /
float(max_x);
      float v = float(j + curand_uniform(&local_rand_state)) /
float(max_y);
      ray r = (*cam)->get_ray(u,v);
      col += color(r, world);
   }
   fb[pixel_index] = col/float(ns);
}
```

Por fim, cada variável terá escrito no vetor fb na posição do pixel processado o resultado final. Devemos lembrar que 'fb' foi declarada usando 'cudaMallocManaged', alocada na Unified Memory, que pode ser acessada tanto da GPU quanto da CPU.



Dessa maneira, podemos, de volta na CPU, acessar o vetor fb e ler os valor processados no kernel render e direcioná-los à saída para gerar a imagem desejada.

```
for (int j = ny-1; j >= 0; j--) {
            for (int i = 0; i < nx; i++) {
                size_t pixel_index = j*nx + i;
                int ir = int(255.99*fb[pixel_index].r()); //pega cor
```

```
vermelha de fb
                int ig = int(255.99*fb[pixel_index].g()); // pega cor
verde de fb
                int ib = int(255.99*fb[pixel_index].b());//pega cor azul
de fb
                std::cout << ir << " " << ig << " " << ib << "\n";
            }
        }
```

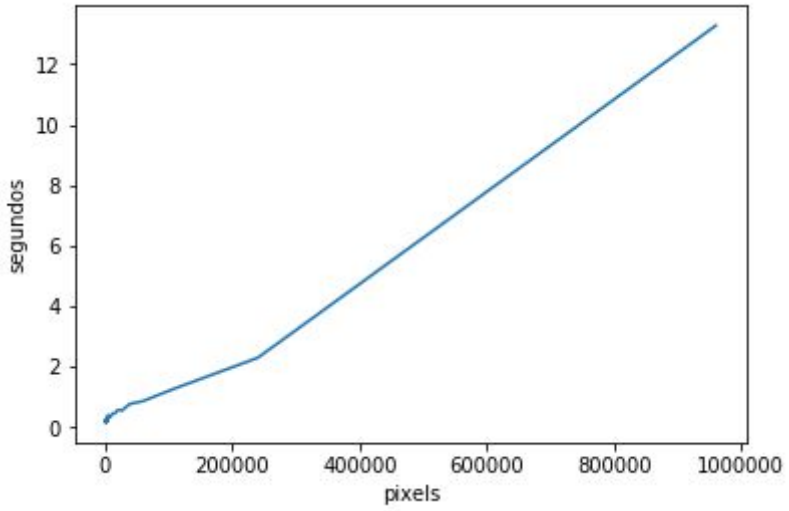
4. Ganhos de Desempenhos

4.1 Testes realizados

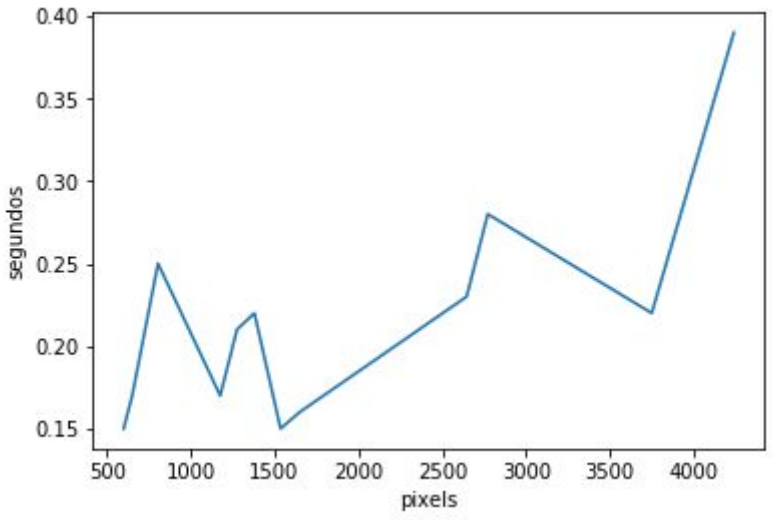
Para observar o ganho de desempenho, realizei um código que realiza o processamento da imagem para diferentes tamanhos e nos indica, ao final do programa, em um arquivo *tempo.txt* o tempo de cada um dos testes.

Tamanho da Imagem	Tempo de Execução (seg)
1200 x 800	14,27
600 x 400	3,29
400 x 266	1,25
300 x 200	0,86
240 x 160	0,76
200 x 133	0,55
171 x 114	0,56
150 x 100	0,45
133 x 88	0,45
120 x 80	0,39
109 x 72	0,33
100 x 66	0,37
92 x 61	0,33

85 x 57	0,28
80 x 53	0,39
75 x 50	0,22
66 x 42	0,28
63 x 42	0,23
50 x 33	0,16
48 x 32	0,15
46 x 30	0,22
44 x 29	0, 21
42 x 28	0,17
35 x 23	0,25
31 x 21	0,17
30 x 20	0,15



Como podemos notar pelos resultados, há um ganho expressivo conforme diminuímos o número de pixel da imagem. No entanto, quando diminuímos muito a imagem chegamos ao seguinte resultado (zoom do gráfico acima):



Nota-se que conforme o número de pixels fica muito pequeno, o tempo não segue mais um modelo linear, pois há um tempo de comunicação entre CPU e GPU que não permitirá que mesmo que seja 1x1 pixel que o tempo de processamento seja de aproximadamente zero segundos.

4.2 Consumo de Memória e Tempo de Execução

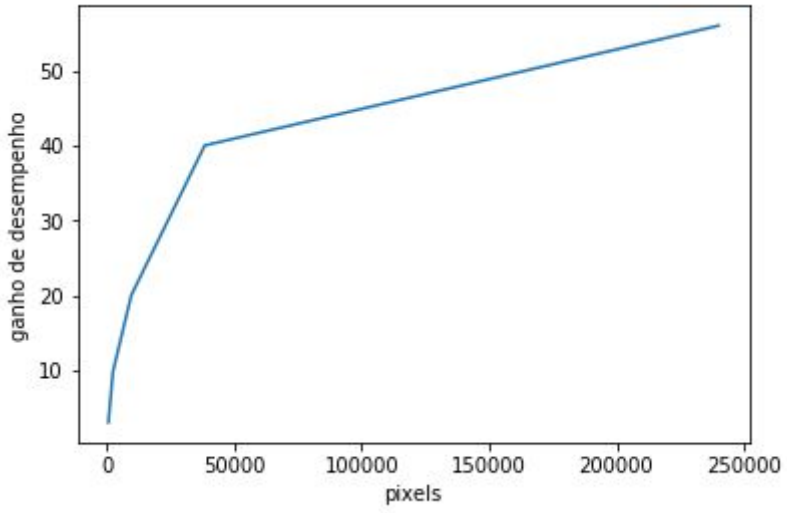
Tamanho da Imagem	Tempo de Execução (seg)	Memória RAM
1200 x 800	14,27	87000
600 x 400	3,29	86944
400 x 266	1,25	86980
240 x 160	0,76	86964
200 x 133	0,55	87044
80 x 53	0,39	87044

Utilizando o comando "top" na linha de comando do linux é possível extrair os dados referente a memória utilizada e o tempo de processo de casa tamanho de imagem. Notamos que não houve 'memory leak' em nenhum teste para nenhuma imagem.

4.3 Comparação com código sem otimização

Tamanho da Imagem	Tempo de Execução Cuda (seg)	Tempo de Execução S/ Otimização (seg)	Ganho com CUDA
600 x 400	3,29	186	56x
240 x 160	0,76	30,06	40x
120 x 80	0,39	7,8	20x
60 x 40	0,20	1,95	9,75x
30 x 20	0,15	0,46	3x

Percebe-se que houve um ganho efetivo com a implementação de Cuda, permitindo ganhos acima de 56x para imagens acima de 600x400. No entanto, nota-se que a proporção de ganho diminui conforme diminui o número de pixel da imagem, em que os processamentos acabam por ficarem quase similares, principalmente pelo tempo de comunicação de API entre GPU E CPU.



4. 4 Como repetir os testes

Para repetir os testes, você deve estar em uma máquina que possua GPU e lá você deve fazer o clone do seguinte repositório do github: https://github.com/antoniosigrist/projeto2-supercomp .

Uma vez com o repositório clonado, você deve executar o comando make para que o código compile. Ele gerará dois arquivos: o image.ppm (imagem final renderizada) e o tempo.txt. O código gerará 40 testes para 40 tamanhos de imagens diferentes e redirecionará o tamanho e tempo de cada teste para o arquivo tempo.txt. Lá você poderá ver os resultados.

Caso haja algum problema ao se roda o make, compile o código com *nvcc main.cu -o main* e rode-o com ./main.

Além disso, está no mesmo repositório o python notebook utilizado para gerar os gráficos do relatório. Você pode alterar os dados com os resultados obtidos no arquivo teste.txt e gerar seus novos gráficos.

Por fim, para realizar os teste comparando com o código não otimizado, recomendo que faça o download do seguinte repositório (https://github.com/petershirley/raytracinginoneweekend). Nele você pode rodar a imagem no tamanho que desejar para ver o tempo de processamento sem processamento paralelo.

Referências:

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

2- https://github.com/petershirley/raytracinginoneweekend

3- https://medium.com/@smallfishbigsea/basic-concepts-in-gpu-computing-3388710e9239

4-https://books.google.com.br/books?id=T8tHDwAAQBAJ&pg=PT290&lpg=PT290&dq=hows+device+and+who%27s+host+gpu+cpu&source=bl&ots=dI1Rm-LceC&sig=ACfU3U12lCGZi1KsBZyntdVGGOH6M5f7VQ&hl=pt-BR&sa=X&ved=2ahUKEwi95du2ob3iAhWzGbkGHRoXCm8Q6AEwC3oECAMQAQ#v=onepage&q=hows%20device%20and%20who's%20host%20gpu%20cpu&f=false