

Trabalho Final - Raytracing em CUDA

Lucas Nogueira Roberto¹ - RA182553

¹Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)

Resumo. Este relatório detalha o desenvolvimento do Trabalho Final da disciplina de Programação Paralela (MC970), ministrada pelo Professor Dr. Hervé Yviquel durante o primeiro semestre de 2024. O projeto envolve a criação de um software de Raytracing, que pode ser executado exclusivamente na CPU com C++ ou utilizando a GPU por meio da linguagem CUDA. Este trabalho aborda o funcionamento da técnica de Raytracing, as diferenças entre as versões seriais e paralelas, um perfilamento básico da versão em CUDA usando o Nsight, e, por fim, uma comparação entre os tempos de execução e o cálculo do speedup do programa em CUDA em relação ao em C++.

1. Introdução

1.1. Raytracing

O algoritmo de raytracing é uma técnica de renderização que simula o comportamento da luz para criar imagens realistas em gráficos de computador. O conceito básico envolve traçar o caminho de raios de luz que viajam do olho do observador através de uma cena 3D para determinar a cor dos pixels na imagem final. Cada raio pode interagir com objetos na cena, refletindo, refratando ou sendo absorvido, o que permite calcular efeitos de sombra, reflexão e refração com alta precisão. Esta técnica resulta em imagens de qualidade excepcional, mas também é computacionalmente intensiva devido ao grande número de cálculos necessários para simular a luz de maneira precisa.

Historicamente, o raytracing foi introduzido nos anos 1960 por Arthur Appel e aperfeiçoado ao longo das décadas seguintes. Nos anos 1980, Turner Whitted expandiu significativamente a técnica ao introduzir a ideia de raios secundários, que permitiu a simulação de efeitos mais complexos como reflexões e transparências. Este avanço marcou o início do uso de raytracing em computação gráfica de alta qualidade, embora a técnica ainda fosse limitada pela capacidade computacional da época. Com o tempo, avanços em hardware e algoritmos permitiram que o raytracing se tornasse mais viável, especialmente com o advento de GPUs potentes.

Nos dias atuais, o raytracing é amplamente utilizado em diversas áreas, desde a indústria cinematográfica até os videogames, graças ao desenvolvimento de hardware especializado e algoritmos otimizados. A introdução de APIs como a NVIDIA RTX e o uso de linguagens como CUDA permitiram que o raytracing fosse acelerado por hardware, tornando possível a renderização em tempo real. Este avanço abriu novas possibilidades para criadores de conteúdo, permitindo a criação de ambientes virtuais mais realistas e imersivos do que nunca. Esse trabalho tem como objetivo explorar essa técnica apartir do zero, entendendo bem ela e implementar uma versão em C++ e em CUDA

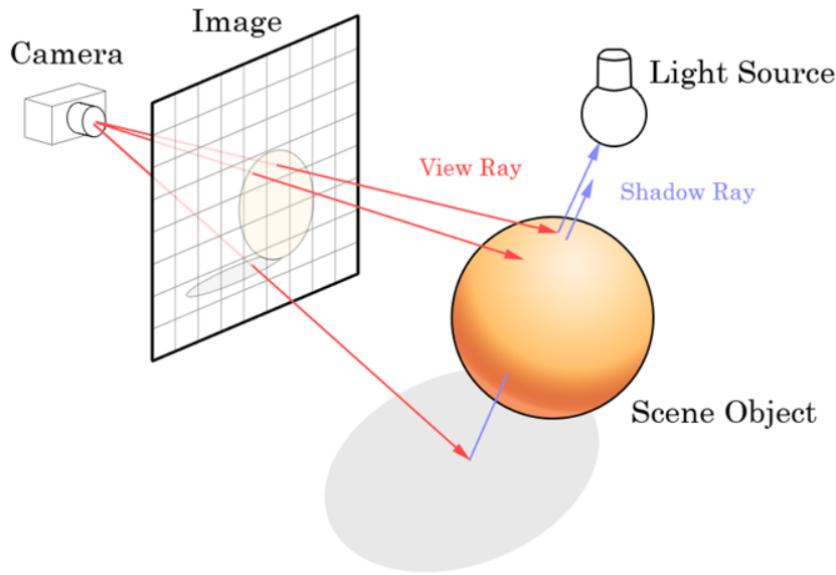


Figura 1. Ilustração sobre o funcionamento da técnica de Raytracing

1.2. CUDA

CUDA, ou Compute Unified Device Architecture, é uma plataforma de computação paralela e uma interface de programação de aplicativos (API) criada pela NVIDIA. Lançada em 2007, a CUDA permite que desenvolvedores utilizem as capacidades de processamento paralelo das GPUs da NVIDIA para tarefas de computação geral, além da renderização gráfica. A principal vantagem da CUDA é permitir que os desenvolvedores escrevam programas usando linguagens familiares, como C, C++ e Fortran, e executem esses programas em milhares de núcleos de GPU em paralelo, proporcionando um aumento significativo no desempenho para aplicações intensivas em dados e cálculos.

Historicamente, antes da introdução da CUDA, as GPUs eram usadas principalmente para gráficos e renderização 3D, e a programação para esses dispositivos era realizada por meio de APIs específicas para gráficos, como OpenGL e DirectX. O lançamento da CUDA representou uma mudança paradigmática, permitindo que as GPUs fossem utilizadas para uma ampla gama de aplicações científicas e de engenharia. A capacidade de executar cálculos em paralelo em larga escala transformou áreas como aprendizado de máquina, simulação física, processamento de imagens e finanças, entre outras. Este avanço foi impulsionado pela crescente demanda por maior poder computacional e a necessidade de processar grandes volumes de dados de maneira eficiente.

O ecossistema CUDA inclui uma variedade de ferramentas e bibliotecas que facilitam o desenvolvimento de aplicações de alto desempenho. O NVIDIA Nsight, por exemplo, é uma ferramenta de desenvolvimento integrada que oferece perfilamento, depuração e análise de desempenho para aplicações CUDA. Além disso, bibliotecas como cuBLAS e cuFFT fornecem implementações altamente otimizadas de operações matemáticas comuns, enquanto o CUDA Toolkit oferece uma coleção abrangente de ferramentas para compilar, depurar e otimizar código CUDA. Com o suporte contínuo da NVIDIA e a adoção crescente na comunidade científica e de engenharia, CUDA continua a ser uma plataforma fundamental para a computação de alto desempenho e inovação tecnológica.

2. Projeto

2.1. Estrutura de Código

Inicialmente, o projeto foi desenvolvido em C. No entanto, à medida que o prazo para a conclusão se aproximava, decidi trocar para C++ na versão da CPU, pois notei que isso proporcionaria menos dificuldades na implementação subsequente da versão em CUDA. Além de que as referências [Peter Shirley 2018] e [Allen 2018] nas quais busquei me basear utilizavam C++ já. A versão antiga do código foi mantida na pasta `_old`. Apesar dessa mudança, toda a parte de criação da janela e grande parte das funções principais do Raytracing foram reaproveitadas.

O projeto agora opera de forma bastante modular. Dentro da pasta `src`, temos os arquivos **main.cpp** e **main.cu**, que são os pontos de entrada para as versões sequencial (CPU) e paralela (GPU), respectivamente. Ambos os arquivos utilizam funções e estruturas de dados presentes nas subpastas **utils**, **window** e **raytracing**.

Cada uma dessas pastas atua como um módulo com responsabilidades específicas, permitindo uma estrutura organizada e eficiente para o desenvolvimento e manutenção do código:

- **Utils:** Contém uma série de funções e macros para facilitar sua gestão e compreensão. No arquivo **types.h**, há um mapeamento dos tipos de dados para que os tamanhos em bits sejam claramente visíveis (por exemplo, `float` → `f32`, `int` → `i32`). No arquivo **utils.h**, encontram-se macros para operações matemáticas, funções utilizadas com frequência e macros de abstração para o código em CUDA como por exemplo as diretivas `DEVICE` e `HOST` que viram `_device_` e `_host_` quando chamadas em um arquivo `.cu` e viram nada quando chamadas em um `.cpp`, ou as funções de geração de números aleatórios específicas para C++ e CUDA, assim esses macros permitem que os códigos na pasta **raytracing** sejam utilizados tanto na versão paralela quanto na versão serial sem grandes mudanças.
- **Window:** Uma adição significativa que implementei e que não encontrei nos tutoriais de raytracing que utilizei como base, foi a criação de uma janela para visualização dos resultados. Para isso, integrei as bibliotecas **glfw** e **glad**, adicionadas na pasta de dependências externas **ext**, e utilizei a API gráfica OpenGL. Assim a pasta **window** contém os códigos e shaders necessários para criar uma janela responsiva. Após a renderização pelo raytracing, os resultados são armazenados em um buffer de pixels, que é um array de *unsigned ints* de 8 bits, e então é gravado em uma textura do OpenGL para exibição na janela criada. Essa abordagem oferece uma visualização interativa dos resultados do raytracing, expandindo significativamente as funcionalidades do projeto. Todo esse comportamento é controlado pela classe **windowContext**, que também é capaz de salvar a imagem em formato `.png` utilizando a única outra dependência externa, a biblioteca **stb_image_write.h**.
- **Raytracing:** Esse diretório contém todo o código da lógica e funcionamento do traçado de raios propriamente dito, ele contém os arquivos **camera.h**, **materials.h**, **worlds.h** e **worlds_cuda.cu** e as subpastas **/geometry** e **/objects**, cada um desses será explicado em seguida começando pelas subpastas:

- **/Geometry:** Contém os arquivos **vec3.h** e **ray.h** que definem as classes de seus respectivos objetos matemáticos. A classe **vec3** encapsula operações vetoriais como adição, subtração, e produtos vetoriais e escalares com vetores em R3, além de funções de amostragem de vetores aleatórios como gerar um vetor 3D qualquer em uma esfera unitária. Já a classe **ray** denota um raio (ou uma semireta) que é composto por 2 **vec3**, o ponto de origem do raio e o vetor de direção dele, essa classe tem a operação de pegar o valor desse raio apartir da parametrização por um valor *t*. Essas classes e suas operações são fundamentais para calcular interseções entre raios de luz e objetos, calcular reflexões e refracções, e para gerar distorções de espalhamento de luz, o que contribui para simulações realistas e imagens bonitas no processo de raytracing.
- **/Objects:** Contém os arquivos **hittable.h** **sphere.h** e **triangle.h**. Em **hittable.h**, a classe **hittable** e sua derivada **hittable_list** gerenciam objetos que podem ser atingidos por raios, em conjunto elas permitem verificar se um raio colide com um objeto específico ou com uma lista de objetos, armazenando informações como o ponto de colisão, a normal da superfície atingida e o material associado. Os arquivos **sphere.h** e **triangle.h** definem os objetos esfera e triângulo, cada um herdando de **hittable** e contendo suas fórmulas de intersecção com raios, a esfera resolvendo uma simples equação quadrática e o triângulo aplicando o algoritmo de Möller-Trumbore.
- **camera.h** Esse arquivo define a classe Camera especificando sua posição, orientação e parâmetros como abertura do diafragma e distância focal. Esses atributos influenciam diretamente a projeção dos raios na cena, afetando a perspectiva e a qualidade da imagem final renderizada.
- **materials.h** Esse arquivo define três tipos de materiais: lambertian, metal, e dielectric. Cada um desses materiais determina como a luz interage com as superfícies dos objetos, controlando propriedades como cor, reflexão e refração. Isso é crucial para a aparência visual das renderizações, oferecendo desde superfícies opacas difusas até reflexões metálicas e transparências dielétricas.
- **worlds.h** e **worlds_cuda.cu** Ambos esses arquivos são responsáveis por agrupar e gerenciar todos os objetos que compõem a cena a ser renderizada, utilizando a classe **hittable_list** para representar uma lista de objetos que podem ser atingidos por raios, e definindo **vec3** como as cores do céu da cena. Foi necessário criar duas versões pois no código em CUDA temos um kernel para a geração do mundo, o que implica em diferentes tipos de parâmetros passados para as funções de criação do mundo em comparação à versão da CPU. Em ambos os arquivos, organizei duas cenas simples com esferas e triângulos. A primeira é a **simple_world**, que foi arquitetada por mim enquanto brincava com o código e a segunda é a **book_cover_world**, que representa a cena da capa do livro que segui

```

src
├── Utils/
│   ├── utils.h
│   ├── logs.h
│   └── types.h
└── Window/
    ├── glfw_window.c
    ├── rendered.c
    ├── callbacks.c
    ├── shaders.c
    └── windowContext.h
Raytracing/
    ├── objects/
    │   ├── hittable.h
    │   ├── sphere.h
    │   └── triangle.h
    ├── geometry/
    │   ├── vec3.h
    │   └── ray.h
    ├── camera.h
    ├── materials.h
    ├── worlds_cuda.cu
    ├── worlds.h
    ├── main.c
    └── main.cu

```

Figura 3. Estrutura do projeto de Raytracing

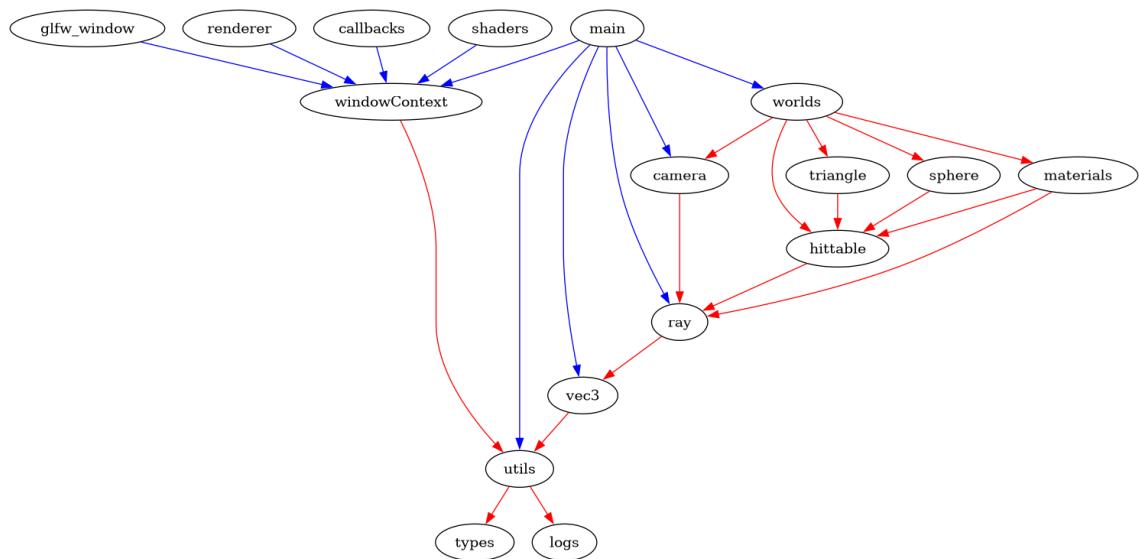


Figura 2. Grafo de dependências entre os arquivos do projeto

2.2. Como executar

Antes de compilar o projeto, é necessário garantir que o computador tenha as seguintes ferramentas instaladas:

- **CMake:** Utilizado para compilar a biblioteca GLFW.
- **Make:** Necessário para rodar o código do projeto utilizando o Makefile.
- **NVCC (NVIDIA CUDA Compiler):** Para compilar o código CUDA.
- **g++:** Para compilar os códigos em C e C++.

Essas ferramentas são importantes para compilar e executar corretamente o projeto, garantindo que todas as dependências sejam atendidas antes da compilação.

Rodar `make all` irá compilar todas as partes necessárias e logo em seguida irá rodar tanto a versão serial quanto paralela. Para rodar alguma parte específica o Makefile oferece as seguintes opções:

- **make glfw:** Este comando cria o diretório de build do GLFW, compila e instala GLFW usando CMake
- **make render:** Compila todos os arquivos C (*.c), C++ (*.cpp) e os arquivos de shaders. Cria um executável main usando g++ e então roda o raytracing na CPU
- **make render_cuda:** Compila todos os arquivos C (*.c) e CUDA (*.cu). Usa nvcc para compilar, linkar e criar um executável main para CUDA e então roda o raytracing na GPU
- **make profile_render_cuda:** Compila e executa o renderizador com CUDA. Permite a análise de desempenho usando NVIDIA Nsight Systems (requer nsys instalado)
- **make clean:** Remove todos os arquivos de build gerados e o executável main.

Para alterar as cenas renderizadas basta alterar as linhas de criação de mundo (world) nos arquivos main.c e/ou main.cu. Para criar as próprias cenas basta editar ou worlds.h ou worlds_cuda.cu dependendo de qual versão do raytracer queira executar.

3. Diferenças entre as versões

Na implementação do ray tracing, grande parte da lógica pode ser compartilhada entre as versões serial (C++) e paralela (CUDA), mas as principais diferenças surgem nos arquivos main.cpp e main.cu onde temos os kernels de CUDA para a renderização de fato. No código em C++, utiliza-se alocação dinâmica com malloc e gerenciamento manual de memória para buffers de textura e estruturas de mundo. Por outro lado, na versão em CUDA, emprega-se cudaMallocManaged e cudaMalloc para alocar e liberar memória de forma controlada na GPU, além de realizar inicializações e cópias de estados do gerador de números aleatórios (curandState) necessárias para o processamento paralelo.

No código em C++, a renderização da imagem pode ser feita de forma que a textura do OpenGL é atualizada com os dados calculados em tempo real, permitindo visualizar a imagem final conforme é gerada, é possível desligar essa funcionalidade para melhor calcular o tempo que esse código toma para criar a cena também. Em contraste, na versão CUDA, não consegui fazer essa funcionalidade de renderização em tempo real funcionar e por isso não foi implementada, mas considero voltar nesse problema no futuro.

Uma outra diferença interessante que apareceu entre as duas versões está na implementação da função rayColor: na versão serial, é utilizada recursão para lidar com reflexões e iluminação indireta, o que pode ser limitado pela capacidade da CPU em lidar com profundidades de recursão muito grandes. Já na versão CUDA, a recursão é substituída por um loop na GPU dentro da função rayColor, aproveitando a paralelização massiva oferecida pelos threads CUDA e evitando problemas de estouro de pilha. As duas implementações podem ser vistas abaixo:

Serial (C++):

```
1 vec3 rayColor(const ray& camera_ray, World* world, i32 depth,
2   ↵ randState* random_state) {
3     if(depth <= 0) return vec3(0, 0, 0);
4
5     // World Objects Collisions
6     hit_record rec;
7     bool hitted = world->collider->hit(camera_ray, 0.001, INF,
8       ↵ rec);
9     if (hitted) {
10       ray scattered_ray;
11       vec3 attenuation;
12       bool scattered = rec.mat_ptr->scatter(camera_ray, rec,
13         ↵ attenuation, scattered_ray, random_state);
14       if(scattered) {
15         return attenuation * rayColor(scattered_ray, world, depth
16           ↵ - 1, random_state);
17       }
18     }
19
20     vec3 unit_direction = normalize(camera_ray.direction());
21     f64 t             = 0.5 * (unit_direction.y() + 1.0);
22     vec3 pixel_color   = (1.0 - t) * world->sky_color1 + t *
23       ↵ world->sky_color2;
```

```
19     return pixel_color;
20 }
```

Paralela (CUDA):

```
1 __device__ vec3 rayColor(const ray &r, World world, curandState
2   ↪ *local_rand_state) {
3   ray cur_ray = r;
4   vec3 cur_attenuation = vec3(1.0, 1.0, 1.0);
5
6   for (i32 i = 0; i < world.ray_max_depth; i++) {
7     hit_record rec;
8     if ((*world.collider))>hit(cur_ray, 0.001f, INF, rec) {
9       ray scattered;
10      vec3 attenuation;
11      if (rec.mat_ptr->scatter(cur_ray, rec, attenuation,
12        ↪ scattered, local_rand_state)) {
13        cur_attenuation = cur_attenuation * attenuation;
14        cur_ray = scattered;
15      } else {
16        return vec3(0.0, 0.0, 0.0);
17      }
18    } else {
19      vec3 unit_direction = normalize(cur_ray.direction());
20      f32 t = 0.5f * (unit_direction.y() + 1.0f);
21      vec3 c = (1.0f - t) * world.sky_color1 + t *
22        ↪ world.sky_color2;
23      return cur_attenuation * c;
24    }
25  }
26  return vec3(0.0, 0.0, 0.0); // exceeded recursion
27 }
```

4. Perfilamento

Para realizar o perfilamento inicialmente considerei utilizar o **nvprof** entretanto aparentemente minha GPU (GeForce RTX 4050) não era suportada mais por ele:

```
~ /p/cp/gp/projeto-final-cuda-raytracing-d/src ➜ P main !6 ?1 ➜ nvprof ./main ✓
===== Warning: nvprof is not supported on devices with compute capability 8.0 and higher.
      Use NVIDIA Nsight Systems for GPU tracing and CPU sampling and NVIDIA Nsight Compute for GPU profiling.
      Refer https://developer.nvidia.com/tools-overview for more details.
```

Figura 4. Mensagem de erro do Nvprof

Então seguindo a recomendação da mensagem de erro fui atrás de utilizar o NVIDIA Nsight para tal, então baixei e instalei ele no meu sistema e adicionei sua chamada no Makefile, gerando o arquivo de perfilamento com **nsys** e o analisando com **nsys-ui**. Para explorar o perfilamento utilizei uma função do nvToolsExt para acompanhar uma parte específica do código também como pode ser observado na timeline do Nsight em rayTrace

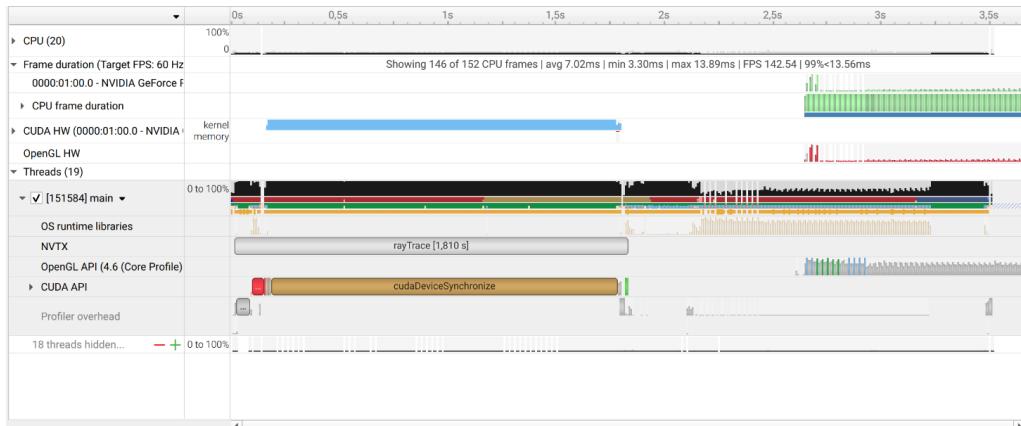


Figura 5. Timeline do Nsight

CUDA API Summary	Time	Total Time	Num Calls	Avg	Med	Min	Max	StdDev	Name
CUDA API Trace	94.0%	1,614 s	5	322,862 ms	356,367 µs	2,813 µs	1,601 s	714,765 ms	cudaDeviceSynchronize
CUDA GPU Kernel Summary	3.0%	62,006 ms	1	62,006 ms	62,006 ms	62,006 ms	0 ns	0 ns	cudaMallocManaged
CUDA GPU MemOps Summary (by Size)	0.0%	14,335 ms	1	14,335 ms	14,335 ms	14,335 ms	0 ns	0 ns	cudaDeviceReset
CUDA GPU MemOps Summary (by Time)	0.0%	12,028 ms	6	2,005 ms	213,137 µs	1,928 µs	10,912 ms	4,371 ms	cudaFree
CUDA GPU Summary (Kernels/MemOps)	0.0%	8,695 ms	5	1,739 ms	49,605 ms	7,694 µs	8,567 ms	3,817 ms	cudaLaunchKernel
CUDA Kernel Trace	0.0%	8,695 ms	5	1,739 ms	49,605 ms	7,694 µs	8,567 ms	3,817 ms	cudaLaunchKernel
CUDA Kernel Launch & Exec Time Trace	0.0%	196,464 µs	5	39,292 µs	2,941 µs	1,485 µs	135,839 µs	58,557 µs	cudaMalloc
CUDA Memory (API/Kernels/MemOps)	0.0%	1,893 µs	1	1,893 µs	1,893 µs	1,893 µs	0 ns	0 ns	cuCtxSynchronize
DX11 PIX Range Summary	0.0%	1,857 µs	1	1,857 µs	1,857 µs	1,857 µs	1,857 µs	0 ns	cuModuleGetLoadingMode
DX12 PIX Range Summary									
MPI Event Summary									
MPI Event Trace									
NVTX GPU Projection Summary									
NVTX GPU Projection Trace									

Figura 6. Sumário da API de CUDA

CUDA API Summary	Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
CUDA API Trace	98.0%	1,601 s	1	1,601 s	1,601 s	1,601 s	0 ns	0 ns	render(vec3 *, int, int, int, Camera **, hitable **, curandStateXORWOW *)
CUDA GPU Kernel/Block Summary	0.0%	12,503 ms	1	12,503 ms	12,503 ms	12,503 ms	0 ns	0 ns	create_world(hitable **, hitable **, Camera **, int, int, curandStateXORWOW *)
CUDA GPU MemOps Summary (by Size)	0.0%	10,904 ms	1	10,904 ms	10,904 ms	10,904 ms	0 ns	0 ns	free_world(hitable **, hitable **, Camera **)
CUDA GPU MemOps Summary (by Time)	0.0%	354,466 µs	1	354,466 µs	354,466 µs	354,466 µs	0 ns	0 ns	render_init(int, int, curandStateXORWOW *)
CUDA GPU Summary (Kernels/MemOps)	0.0%	1,248 µs	1	1,248 µs	1,248 µs	1,248 µs	0 ns	0 ns	rand_init(curandStateXORWOW *)
CUDA Kernel Trace									
CUDA Kernel Launch & Exec Time Trace									
CUDA Memory (API/Kernels/MemOps)									
DX11 PIX Range Summary									
DX12 GPU Command List PIX Ranges Summa									
DX12 PIX Range Summary									
MPI Event Summary									
MPI Event Trace									
NVTX GPU Projection Summary									
NVTX GPU Projection Trace									

Figura 7. Sumário dos kernels do código

5. Resultados

5.1. Qualitativos (Imagens):

Abaixo pode-se observar algumas das imagens que gerei com esse raytracer. Em algumas temos os parâmetros *Samples per pixel* que indica quantos raios de luz foram amostrados para calcular a cor de cada pixel e *Ray Max Depth* que é o limite da recursão na função RayColor.

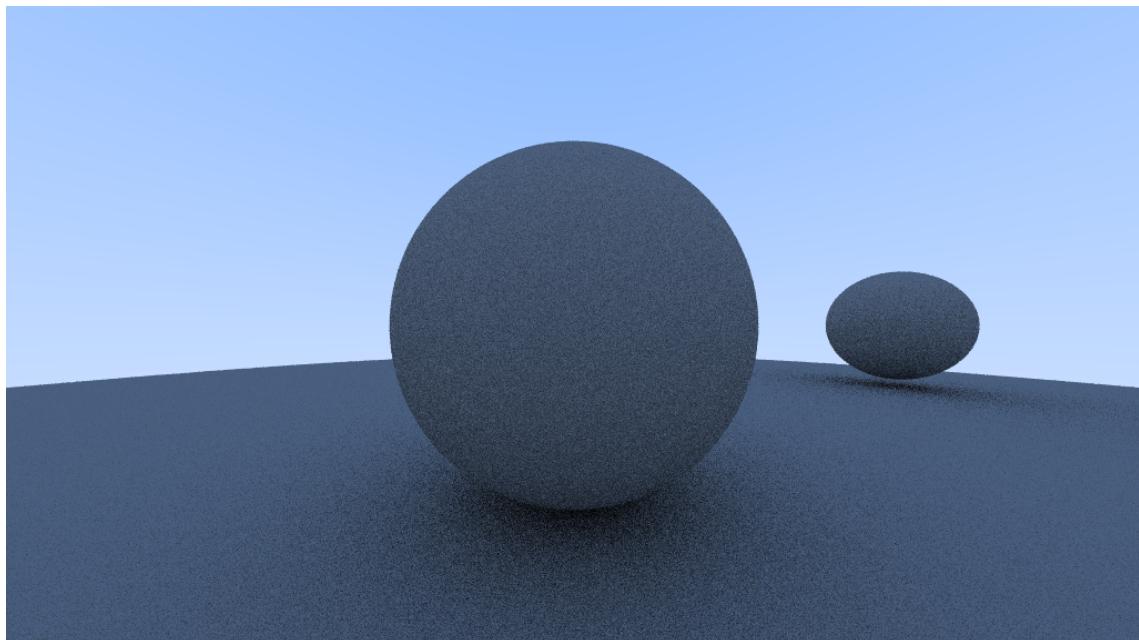


Figura 8. Uma das primeiras imagens obtidas

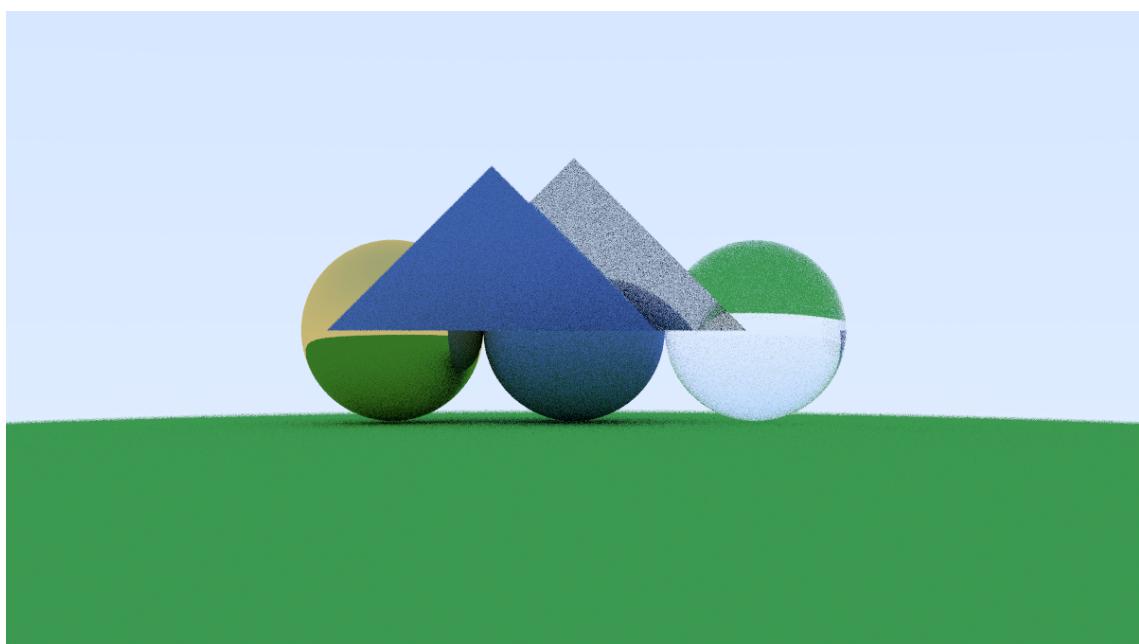


Figura 9. Cena simple world — Samples per pixel: 10 — Ray Max Depth: 20

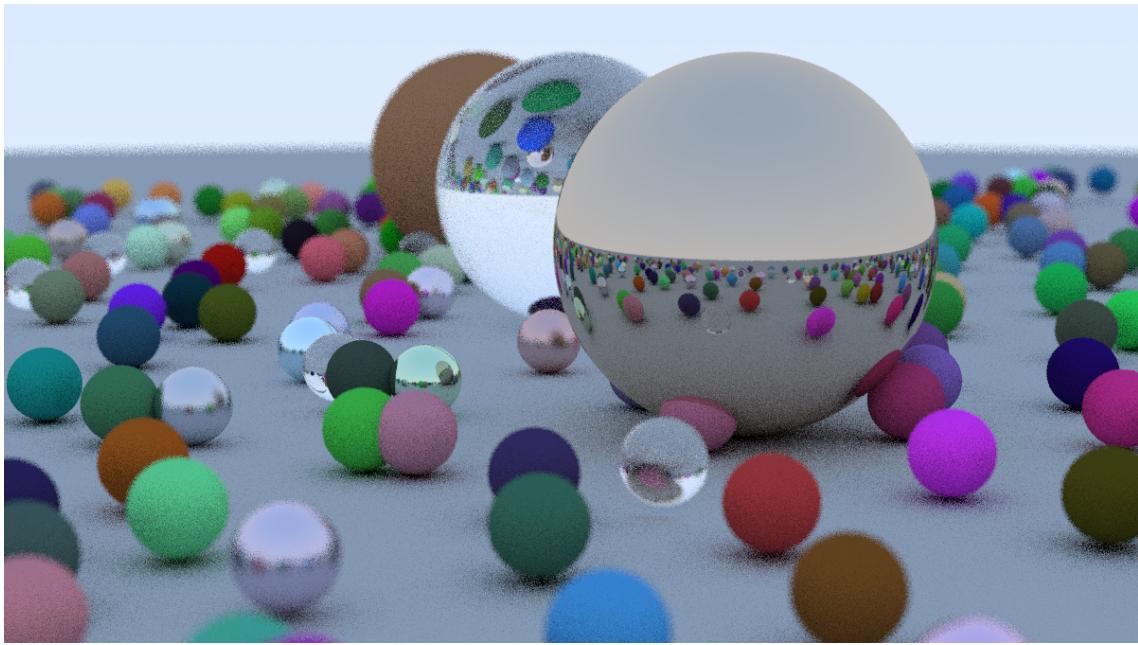


Figura 10. Cena book cover — Samples per pixel: 10 — Ray Max Depth: 20

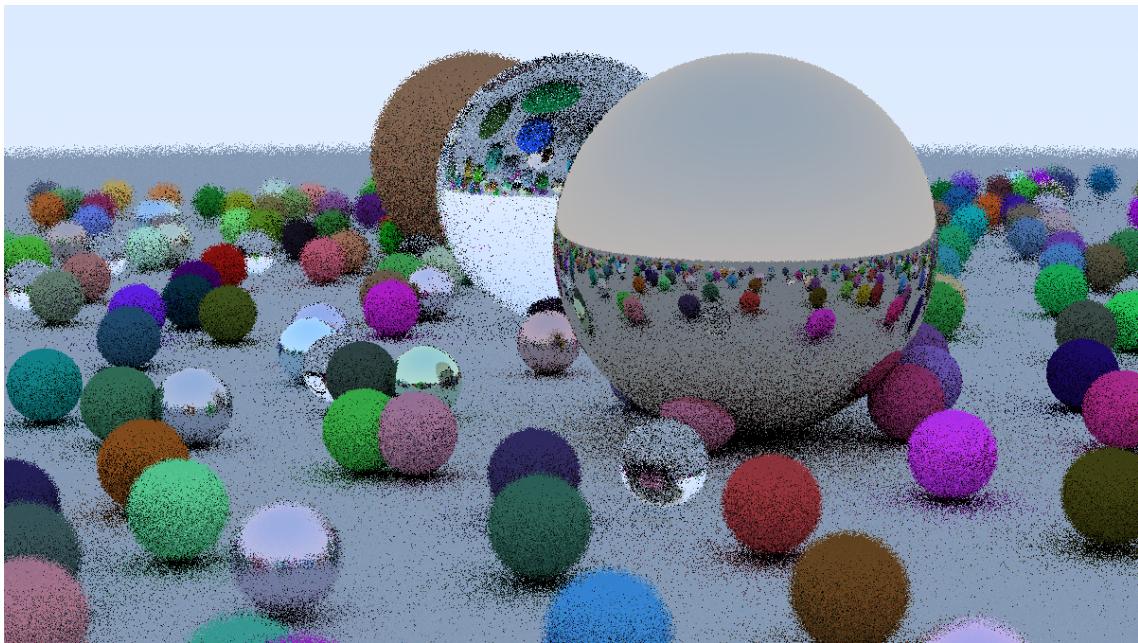


Figura 11. Cena book cover — Samples per pixel: 1 — Ray Max Depth: 5

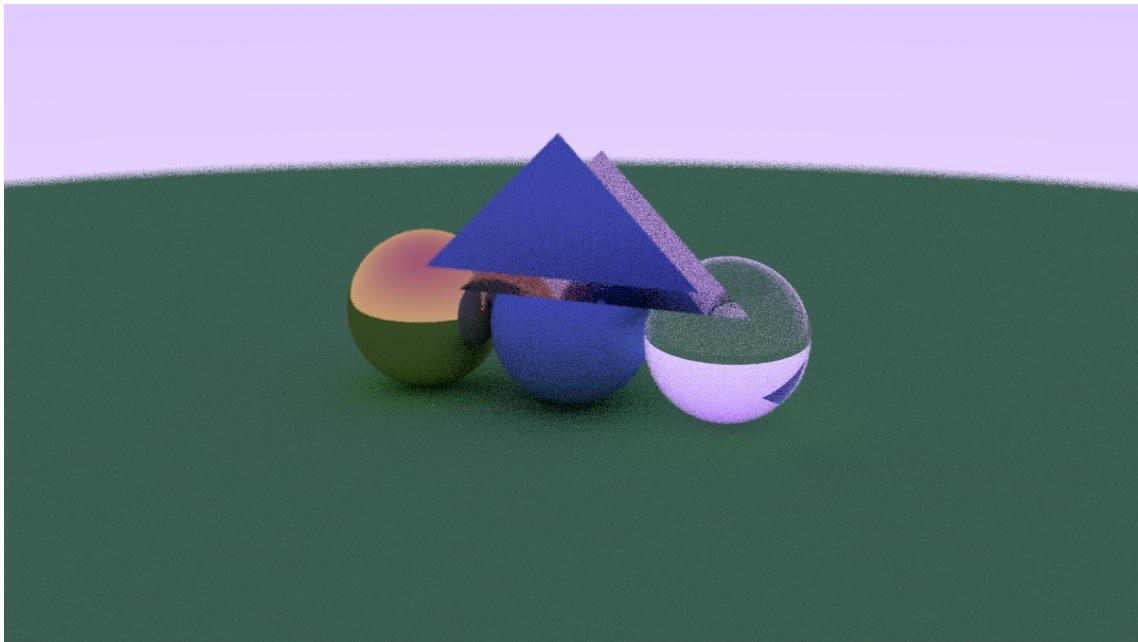


Figura 12. Cena simple world — Samples per pixel: 20 — Ray Max Depth: 20 — Outra posição de Câmera e cor de céu

5.2. Quantitativos (Speedup):

Abaixo estão as configurações do computador utilizado para efetuar o benchmark do código paralelo e serial. Logo em seguida tem as tabelas comparando os Speedups em diferentes cenários.

Platform	Linux
OS	Manjaro Linux
Hardware platform	x86_64
Serial number	Local (CLI)
CPU description	13th Gen Intel(R) Core(TM) i7-13650HX
GPU descriptions	NVIDIA GeForce RTX 4050 Laptop GPU
NVIDIA driver version	550.54.14
Max EMC frequency	1.60 GHz
CPU context switch	supported
GPU context switch	supported
Guest VM id	0
Tunnel traffic through SSH	no
Timestamp counter	supported
Profiling session UUID	be80f2b3-b83d-49fa-b898-0dc44fde9db3
Target name	turing

Tabela 1. Informação do sistema utilizado

Os tempos obtidos foram uma média de 10 experimentos em cada uma das versões.

Cena	C++	CUDA	Speedup
Simple World	1.09s	0.005s	218x
Book Cover World	19.13	0.212	90x

Tabela 2. Tabela de Speedups

Imagen: 1200 x 675

Samples per pixel: 1

Ray Max Depth: 20

Cena	C++	CUDA	Speedup
Simple World	10.62s	0.032s	332x
Book Cover World	202.95s	1.841s	111x

Tabela 3. Tabela de Speedups

Imagen: 1200 x 675

Samples per pixel: 10

Ray Max Depth: 20

Cena	C++	CUDA	Speedup
Simple World	21.24s	0.041s	518x
Book Cover World	381.45s	3.70s	103x

Tabela 4. Tabela de Speedups

Imagen: 1200 x 675

Samples per pixel: 20

Ray Max Depth: 20

Cena	C++	CUDA	Speedup
Simple World	9.64s	0.024s	402x
Book Cover World	178.98s	1.344s	133x

Tabela 5. Tabela de Speedups

Imagen: 1200 x 675

Samples per pixel: 10

Ray Max Depth: 5

Cena	C++	CUDA	Speedup
Simple World	2.64s	0.009s	293x
Book Cover World	50.28s	0.534s	94x

Tabela 6. Tabela de Speedups

Imagen: 600 x 675

Samples per pixel: 10

Ray Max Depth: 20

6. Conclusão

Neste projeto, foi desenvolvido um software de Ray Tracing utilizando OpenGL, C++ e CUDA, capaz de renderizar diferentes cenas compostas por esferas e triângulos, formados por diversos tipos de materiais. Durante o desenvolvimento, foram exploradas técnicas fundamentais de computação gráfica e computação paralela, resultando em imagens de alta qualidade.

A organização modular do código permitiu uma separação clara das responsabilidades entre renderização, geometria, interação de raios com objetos e manipulação de materiais. A utilização de OpenGL forneceu uma base sólida para a renderização, enquanto o CUDA acelerou o processo através da paralelização em GPUs, resultando em uma melhoria significativa no desempenho e na complexidade das cenas renderizadas.

Em comparação com a versão serial em C++, a versão paralela mostrou-se centenas de vezes mais rápida, alcançando speedups significativos. Uma observação interessante é que o aumento de desempenho em cenas simples foi muito mais rápido do que em cenas complexas ao se aumentar o número de amostras de raios. Isso indica que, à medida que a cena se torna mais complexa, até mesmo a versão em CUDA enfrenta dificuldades para renderizar tudo em um tempo curto.

Além disso, a experiência adquirida com este projeto não apenas aprimorou minhas habilidades técnicas em programação gráfica e computação paralela, mas também expandiu meu entendimento sobre os princípios teóricos subjacentes ao Ray Tracing. Estou satisfeito com os resultados obtidos e pretendo continuar aprimorando este ray tracer como um projeto pessoal, com o objetivo de alcançar uma renderização em tempo real utilizando CUDA.

Referências

- Allen, R. (2018). Accelerated ray tracing in one weekend in cuda.
NVIDIA Corporation (2024). *NVIDIA Nsight Documentation*. Accessed: 2024-06-26.
Peter Shirley, Trevor David Black, S. H. (2018). Ray tracing in one weekend.