Insper - Engenharia de Computação - 7° Semestre

Projeto 2
Ray Tracing em CUDA

Bruna Mayumi Kimura

Supercomputação - Prof. Luciano Soares

1. Introdução

O objetivo deste trabalho é otimizar um código, utilizando para tanto uma GPU. Para o programação da GPU foi utilizado o CUDA em C++. O código escolhido para ser otimizado é um programa de ray tracing. O projeto foi totalmente baseado no artigo “Ray Tracing in one weekend” de Peter Shirley[1], e foram feitas apenas algumas alterações para a otimização deste código. O código pode ser encontrado inteiramente no GitHub do mesmo “petershirley/raytracinginoneweekend”[2].

Para analisar os ganhos e perdas de desempenho foi utilizado um contador de tempo, a biblioteca “*time”*, do próprio C++. O computador utilizado é uma vm da amazon k80.

2. Metodologia

2.1. Algoritmo e código

O projeto foi feito em C++. Foi utilizada as bibliotecas padrão *std* e as funções do CUDA para as otimizações do código.

O projeto possui um arquivo principal, denominado *main.cpp*, responsável por gerar a imagem final. Além disso, há outros 5 arquivos header. O *vec3.h* são as funções de sobrecarga de operadores. Permitindo realizar operações entre vetores de forma mais simples. Já o *ray.h* é responsável por gerar os raios do *ray tracing*, todos os tipos de técnicas de *ray tracing*, necessitam desta classe. A próxima classe a ser criada foi a *sphere.h*, como o nome já diz, é responsável por criar as esferas na imagem. A classe *hitable.h* e *hitable_list.h* calcula os raios que atingem a esfera. Já o código da *camera.h* é responsável pelo posicionamento da imagem. Por fim, o *material.h* é responsável por mudar o tipo de material que é feita as esferas. Para maior entendimento do algoritmo é necessário ler o artigo “Ray Tracing in one weekend”.

Nesse projeto de GPU, foram implementados apenas alguns dos passos, ou seja, o projeto não foi passado inteiramente para a GPU. Esse projeto foi feito até o passo 5 do artigo, porém adicionado duas esferas ao invés de 1.

Primeiramente é necessário compilar o programa. Para que ele execute normalmente é necessário que o computador utilizado possua GPU. Assim, basta executar o comando da **Figura 1,** ou simplesmente digitar "*make*" no terminal. Já para gerar a imagem basta redirecionar a saída do programa para um arquivo *.ppm*, como mostra a **Figura 2**.

```
nvcc main.cu -o main
```

Figura 1: compilando a main na GPU

```
./main >> imagem.ppm
```

Figura 2: redirecionamento da imagem para arquivo *imagem.ppm*

2.2. Gerando o cenário

O primeiro passo do projeto é criar o cenário da imagem. Para tanto é necessário apenas gerar uma imagem 800x600 colorida.

```
#include <iostream>

int main() {
    int nx = 200;
    int ny = 100;
    std::cout << "P3\n" << nx << " " << ny << "\n255\n";
    for (int j = ny-1; j >= 0; j--) {
       for (int i = 0; i < nx; i++) {
           float r = float(i) / float(nx);
           float g = float(j) / float(ny);
           float b = 0.2;
           int ir = int(255.99*r);
           int ig = int(255.99*g);
           int ib = int(255.99*b);
           std::cout << ir << " " << ig << " " << ib << "\n";
       }
    }
}
```

Figura 3: código original do ray tracing, apenas gerando uma imagem simples

A **Figura 3** representa o código original retirada do artigo base. Para passar para a GPU o caso base, o primeiro passo é decidir o que será enviado para a mesma para fim de agilizar os cálculos. Nesse caso, o que se quer passar para a GPU calcular são todas as declarações dentro dos *for*'s.

Primeiramente foi necessário definir o tamanho da imagem no eixo x e y (NX e NY respectivamente) como mostra a **Figura 4**. Além de definir o tamanho *SIZE*, que será utilizado para alocação de memória da matriz da imagem, tanto na CPU quanto na GPU.

```
#define NX 800
#define NY 400
#define SIZE NX*NY*3*sizeof(int)
```

Figura 4: Definição da imagem e tamanho

Após as definições, agora dentro da main, foi necessário criar a o grid e o block, além de alocar duas matrizes, uma para CPU e outra para GPU. A **Figura 5** mostra como ficou o código.

```
int main(){

    dim3 dimGrid(ceil(NX/(float)16), ceil(NY/(float)16));
    dim3 dimBlock(16, 16);

    int *cpu_matrix;
    int *gpu_matrix;

    cpu_matrix = (int *)malloc(SIZE);
    cudaMalloc((void **)&gpu_matrix,SIZE);
```

Figura 5: Grid, Block e matrizes devidamente alocadas

Todo os passos dos *for's* agora estarão em um kernel na GPU. Os índices *i* e *j* são dados pela expressão: `(blockIdx.x* blockDim.x + threadIdx.x)` e `(blockIdx.y* blockDim.y + threadIdx.y)` respectivamente.

Já para preencher a matriz a **Figura 6** mostra como ficou o código. O restante do código permanece igual, porém agora dentro da kernel.

```
matrix[(i*NY + j)*3] = ir;
matrix[(i*NY + j)*3 + 1] = ig;
matrix[(i*NY + j)*3 + 2] = ib;
```

Figura 6: Preenchendo a matriz da GPU

Voltando para a main, agora o objetivo é mandar os dados para a função do kernel. Assim, é necessário configurar como mostra a **Figura 7**. A função do kernel chama-se *generate*. Após fazer os cálculos na GPU, é necessário recuperar esses dados, para tanto foi utilizado a função *cudaMemcpy*, que escreve na matriz da CPU os resultados obtidos. Por fim, após a cópia de dados é necessário liberar o espaço de memória da GPU que não está sendo mais usado, para isso, foi utilizado o comando *cudaFree*.

```
generate<<<dimGrid, dimBlock>>>(gpu_matrix)

cudaMemcpy(cpu_matrix, gpu_matrix, SIZE, cudaMemcpyDeviceToHost);
cudaFree(gpu_matrix);
```

Figura 7: mandar matriz para kernel, copiar dados para a GPU e liberar o espaço de memória

O resultado da final da imagem pode ser vista na **FIgura 8**.

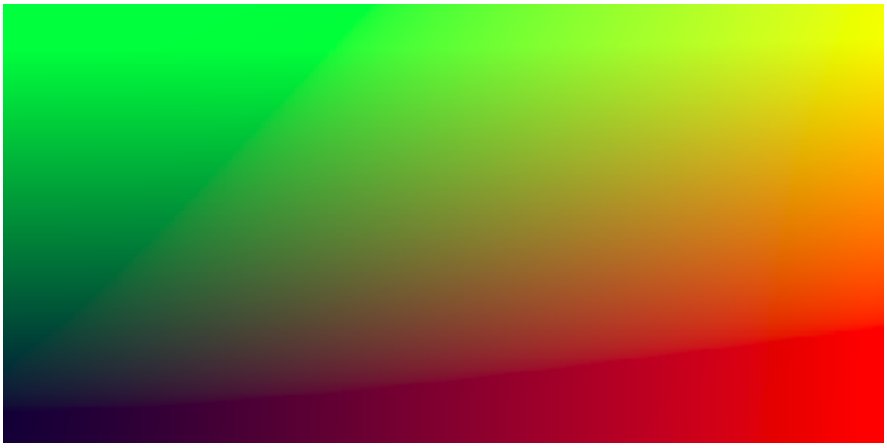

Figura 8: Resultado final da imagem da primeira iteração

2.3. Adicionando vetores

O segundo passo é adicionar os vetores vec3 para gerar a mesma imagem. Para tanto foi necessário adicionar o arquivo *vec3.h*. Todas as funções desse arquivo foram enviadas para a GPU. A **Figura 9** mostra como ficou essa situação.

```
class vec3  {


public:
    __host__ __device__ vec3() {}
    __host__ __device__ vec3(float e0, float e1, float e2) { e[0
    __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 { r
    __host__ __device__ inline vec3 operator-() const { return v
    __host__ __device__ inline float operator[](int i) const { r
    __host__ __device__ inline float& operator[](int i) { return

    __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);
```

Figura 9: Classe *vec3* enviado para GPU

Para gerar a imagem nesse caso, não precisa fazer nenhuma alteração. Basta compilar e redirecionar a saída da mesma forma que mostrado no item **2.1**. Por fim,a sáda está mostrada na **Figura 10**.

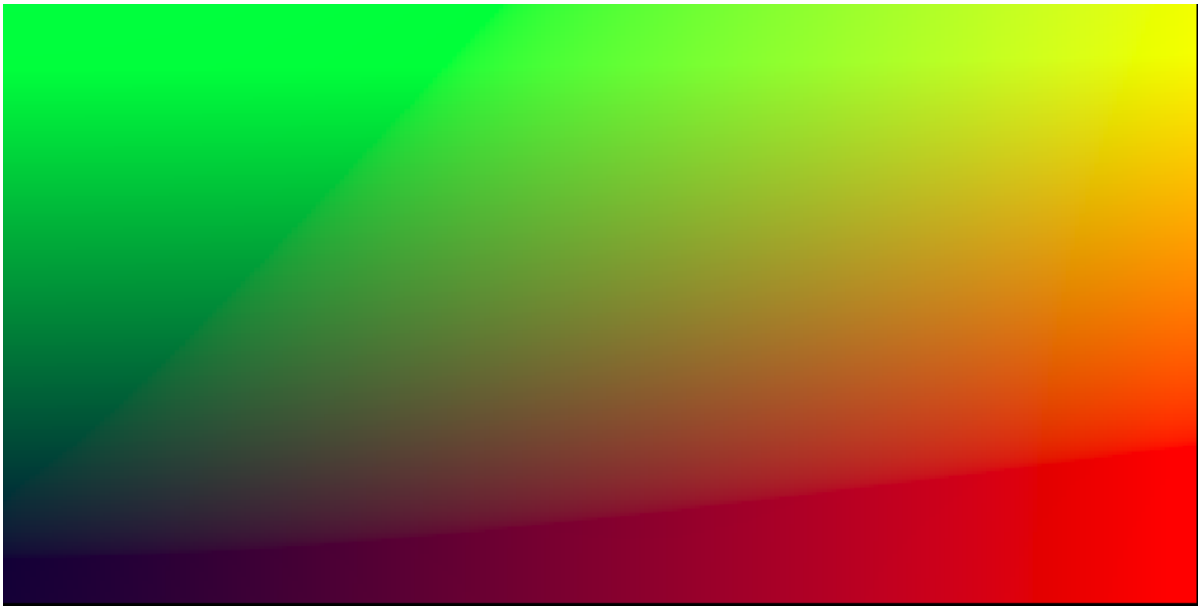

Figura 10: Fundo gerado utilizando a classe *vec3*

Após isso, foi gerada uma imagem em degradê de azul utilizando os raios. Para tanto foi necessário adicionar a classe *ray* no projeto. O resultado da imagem pode ser vista na **Figura 11**.

Figura 11: Fundo gerado utilizando a classe *ray*

2.4. Apenas uma esfera

Para gerar uma esfera simples em cima do fundo da **Figura 11**, foi utilizada uma função chamada *hit_sphere*. A **Figura 12** representa essa função. Como pode ser visto essa função também foi enviada para GPU.

```
__device__ bool hit_sphere(const vec3& center, float radius, const ray& r){
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = 2.0 * dot(oc, r.direction());
    float c = dot(oc, oc) - radius*radius;
    float discriminant = b*b - 4*a*c;
    return(discriminant > 0);
}
```

Figura 12: Código que gera a esferal

A imagem gerada no final é uma esfera vermelha no fundo azul. A **Figura 13** representa essa situação.

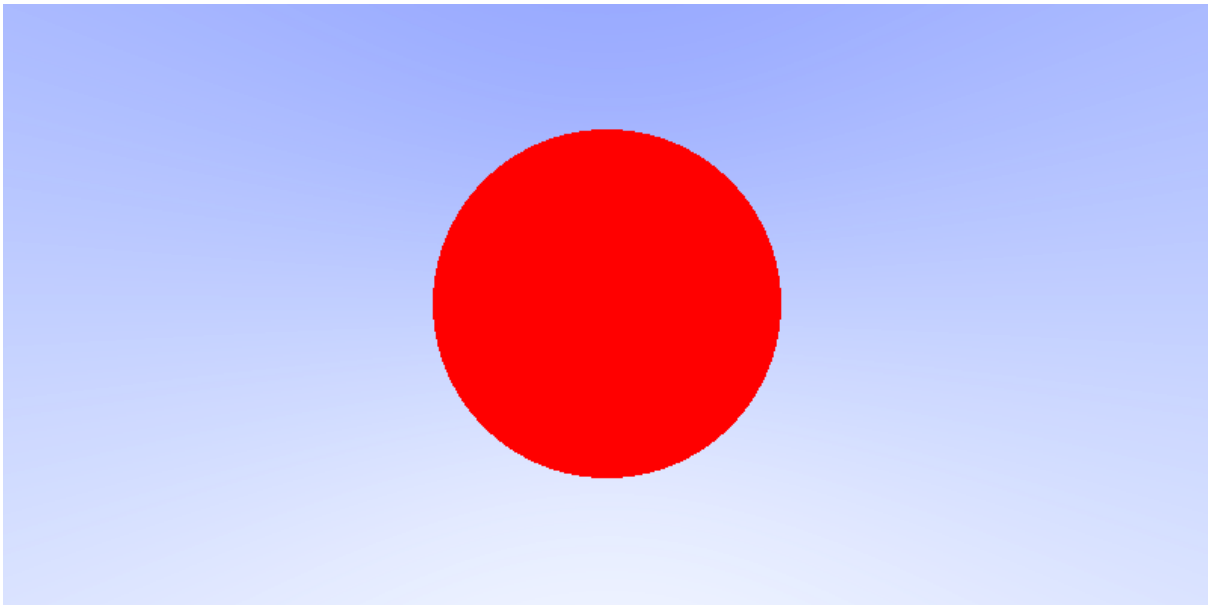

Figura 13: Esfera vermelha no fundo azul

O próximo passo foi fazer a mesma esfera, porém agora com alguma iluminação. Para tanto foi necessário alterar o código do da função do *sphere_hit* e *color*, como mostra o artigo base desse projeto. O resultado final dessa alteração pode ser vista na **Figura 14**.

Figura 14: Esfera com iluminaçãol

2.5. Duas esferas

Para gerar mais de uma esfera, é necessário fazer as seguintes classes: *sphere.h*, *hitable.h, hitable_list.h*. Todos essas classes foram enviadas a GPU com a adição do "*__device__*" na frente de cada função. Essa parte do projeto teve ajuda do projeto do site CUDA que possui o mesmo propósito de otimização deste código na GPU[3]. Uma parte dessa parte do projeto (criação do "*create_world*" e "*free_world*") foi baseada nesse GitHub[4].

O primeiro passo foi alocar espaço na memória da CPU e GPU para a geração das esferas. A **Figura 15** mostra como ficou o código. Para a criação da imagem foi necessário criar dois kernel's a mais, um para criar as esferas e outro para deletar o espaço na memória. Esses kernel's podem ser vistos na **Figura 16**. Já a chamada do kernel de criação pode ser vista na última linha da figura anterior (**Figura 15**).

```
hitable **d_list;
cudaMalloc((void **)&d_list, 2*sizeof(hitable *));
hitable **d_world;
cudaMalloc((void **)&d_world, sizeof(hitable *));
create_world<<<1,1>>>(d_list,d_world);
```

Figura 15: Alocando espaço na GPU para as esferas

```
__global__ void create_world(hitable **d_list, hitable **d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        *(d_list)   = new sphere(vec3(0,0,-1), 0.5);
        *(d_list+1) = new sphere(vec3(0,-100.5,-1), 100);
        *d_world    = new hitable_list(d_list,2);
    }
}

__global__ void free_world(hitable **d_list, hitable **d_world) {
    delete *(d_list);
    delete *(d_list+1);
    delete *d_world;
}
```

Figura 16: Kernel para criação e liberação do espaço de memória

O restante do código do arquivo principal também sofre pequenas alterações como pode ser conferido no artigo base desse projeto[1], capítulo 5.

Assim, como saída dessa iteração temos a **Figura 17.**

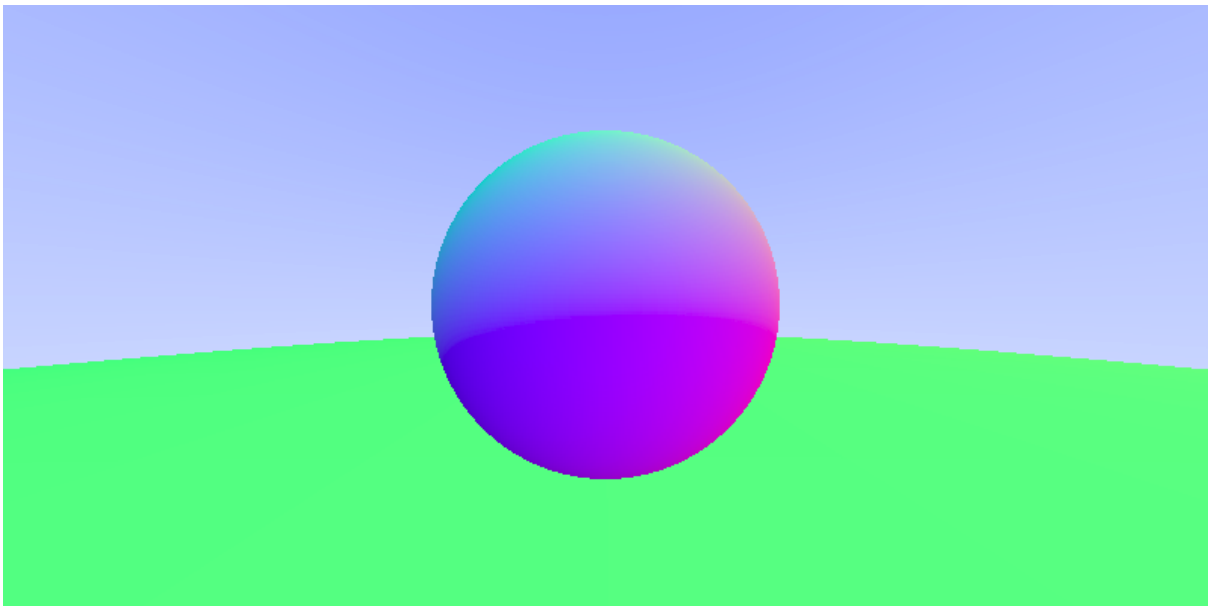

Figura 17: Imagem final do capítulo 5 do artigo

Por fim, foi adicionado mais duas esferas como meio de teste. Foi com esse código final que foram feitas todas os itens do item 3, deste relatório. A **Figura 18** mostra como ficou o resultado final.

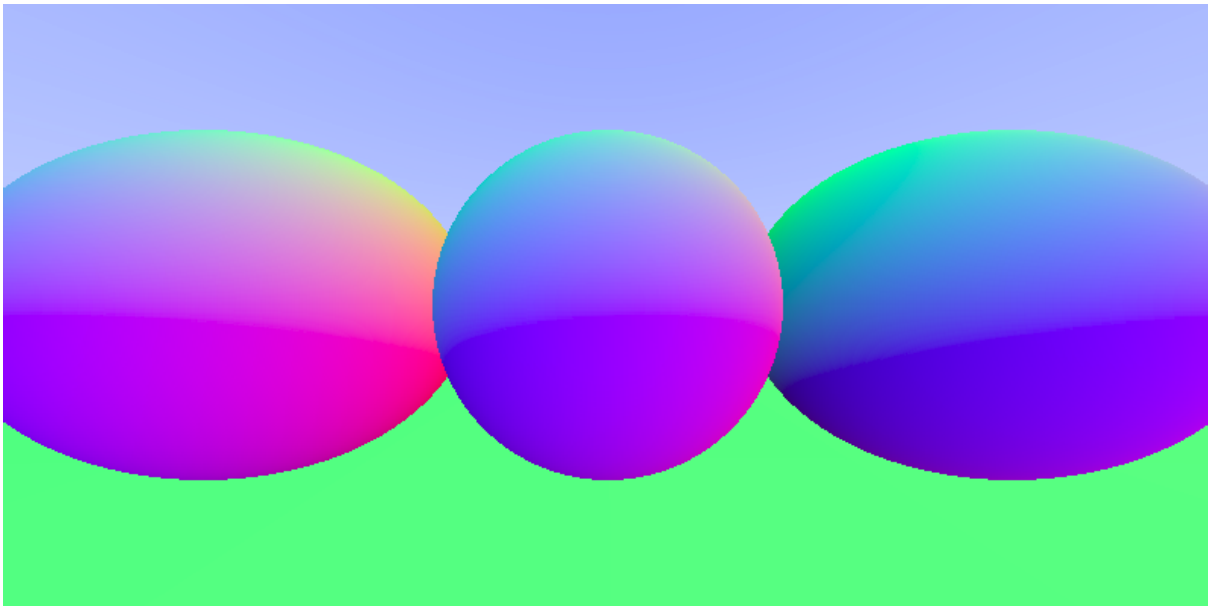

Figura 18: Imagem final com 4 esferas

3. Resultados

3.1. Comparação entre tempos de GPU e CPU

Depois de ter gerado o código na GPU para quatro esferas (a quarta esfera é o chão verde) foi necessário refazer o mesmo código para CPU.

O arquivo com as funções CUDA foi executado na máquina descrita na introdução. Já o arquivo sem CUDA foi executada em máquina local, com processador i5 e 4 GB de RAM.

O programa foi executado para oito tamanhos diferentes de resolução de imagem. A **Tabela 1** representa os tempos obtidos na GPU e CPU. Para a medição do tempo, o programa começa a contar o tempo a partir da execução da main, e termina na finalização da mesma. Essa situação para ambos os casos.

Tabela 1: Tabela de Tempo de execução para várias resoluções

	CPU	GPU
200x100	0,028104 seg	0,746432 seg
400x200	0,098435 seg	0,770335 seg
800x400	0,374885 seg	0,892205 seg
1200x800	1,26119 seg	1,06846 seg
2400x1200	3,89032 seg	2,21803 seg
9600x4800	57,3952 seg	17,264 seg
12800x6400	106,141 seg	32,7135 seg
25600x12800	403,671 seg	149,122 seg

Além disso, foi gerado um gráfico com as informações acima. Para melhor visualização os dados estão em escala logarítmica. o **Gráfico 1**, representa os dados.

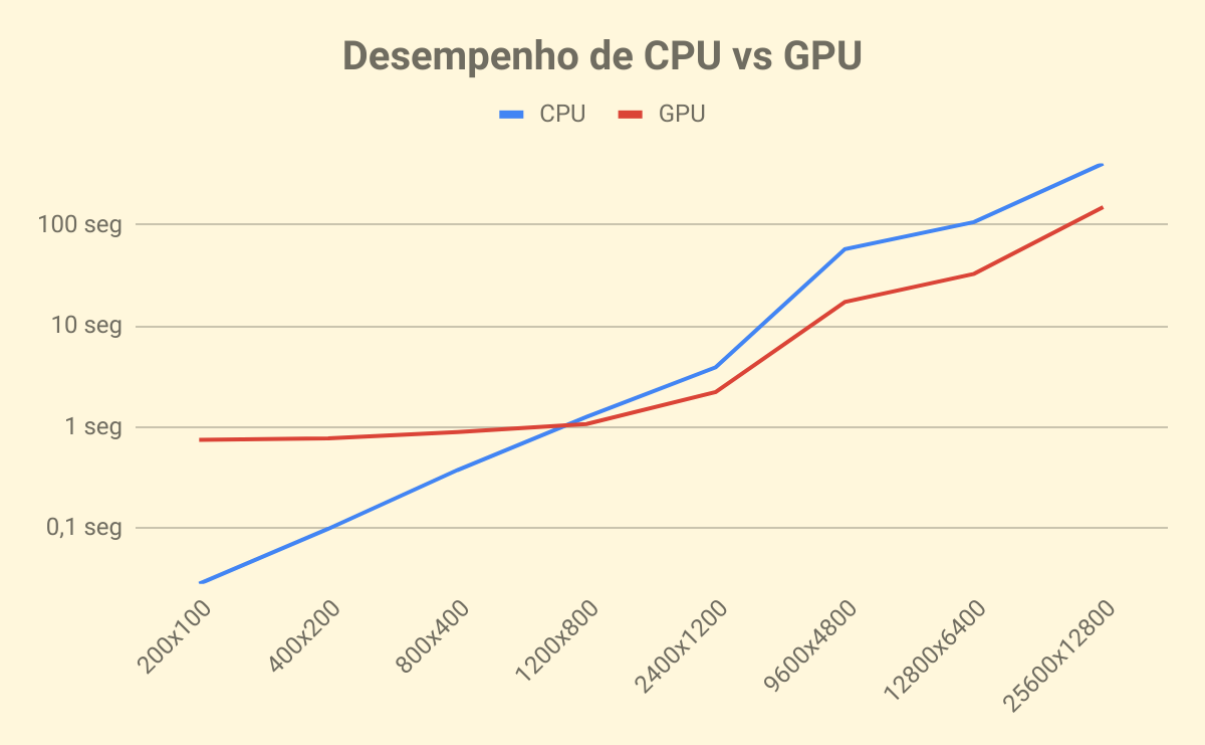

Gráfico 1: Gráfico do desempenho da CPU em comparação com o da GPU

4. Conclusão

No fim, foi possível notar que para imagens pequenas o desempenho da CPU é bem melhor que a da GPU. Como a medição de tempo se inicia logo no início da função *main* é de se esperar tal resultado, pois a alocação de memória é feita dentro dessa contagem de tempo. Ou seja, provavelmente para matrizes muito pequena o desempenho da GPU é pior por estar fazendo as alocações na memória da GPU e trocando os dados entre elas. Isso é um fator que prejudica o desempenho da GPU, pois nesse caso, o tempo de alocação é maior que o de execução.

Conforme a resolução aumenta, ou seja, o tamanho da matriz aumenta, é possível ver que o tempo de execução da GPU é bem menor que da CPU. No último caso analisado, por exemplo, a matriz tem tamanho 25,6 mil por 12,8 mil, nesse caso o tempo de execução da GPU foi mais de 4 minutos mais rápida que a execução da CPU. Essa situação apenas confirma a suposição feita acima, em que o fato da CPU ter executado mais rápido o programa para matrizes pequenas é justamente pelo fato da alocação de memória na GPU ser mais lenta.

5. Bibliografia

[1] SHIRLEY, Peter. Ray Tracing in One Weekend. Version 1.55. Disponível em: <https://drive.google.com/drive/folders/14yayBb9XiL16lmuhbYhhvea8mKUUK77W>. Último acesso em: 5 de Junho de 2019.

[2] ray tracing in one weekend, GitHub. petershirley/raytracinginoneweekend. Disponível em: <https://github.com/petershirley/raytracinginoneweekend>. Último acesso em: 5 de Junho de 2019.

[3] ALLEN, Roger. Accelerated Ray Tracing in One Weekend in CUDA. NVIDIA Developer Blog. 5 de Novembro de 2018. Disponível em:<https://devblogs.nvidia.com/accelerated-ray-tracing-cuda/>.Último acesso em: 5 de Junho de 2019.

[4]ray tracing in one weekend in cuda, GitHub. rogerallen/raytracinginoneweekendincuda. Disponível em: <https://github.com/rogerallen/raytracinginoneweekendincuda>. Último acesso em: 5 de Junho de 2019.