Relatório Projeto 1

Antonio Lacombe Sigrist

São Paulo, 02 de abril de 2019

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

2. Otimização Realizadas

As otimizações abaixo realizadas foram feitas e testadas em um MacBook Air 2,2 GHz Intel Core i7. A primeira otimização realizada no código foi adicionar um #pragma omp parallel for para paralelizar o for entre as threads sendo realizado no interior da main. Essa otimização foi responsável em melhorar o desempenho em **44%,** aproximadamente:

```
for (int j = ny-1; j >= 0; j--) {

    for (int i = 0; i < nx; i++) {
        vec3 col(0, 0, 0);
        #pragma omp parallel for
        for (int s=0; s < ns; s++) {

            float u,v;

            u = float(i + drand48()) / float(nx);
            v = float(j + drand48()) / float(ny);
            ray r = cam.get_ray(u, v);
            vec3 p = r.point_at_parameter(2.0);
            vec3 color2 = color(r, world,0);
```

No entanto, apenas paralelizar esse for faz com que diversas threads tentem escrever ao mesmo tempo na variável **col,** o que pode gerar micro erros na imagem, mesmo que imperceptíveis aos olhos. Para contornar esse problema, adicionamos temos que proteger a variável col, e poderíamos fazer isso com um atomic, um critical ou um recution.

Como a variável col é do tipo vec3, o reduction deveria ser repensado, quebrando a variável em outras que pudessem ser usadas pela função reduction. Caso usássemos o atomic, ele protegeria o col como que fazendo que apenas uma thread escrevesse no col como se fosse uma thread so rodando. Como o grosso do processamento do código está rodando ali, o tempo de processamento sobe quase como se não tivesse paralelismo.

Dessa maneira, a forma escolhida para contornar o problema foi usando o critical. Se apenas adicionássemos #pragma omp critical, ele protegeria tanto o col quanto a chamada de função **color(r, world,0).** Assim, chamamos fazemos uma mudança de variável atribuindo a color() a uma outra variável e apenas chamando a variável dentro do critical para evitar que ela seja protegida. Essa solução, combinada com o for, traz um desempenho de **48%,** aproximadamente, além de tornar o resultado consistente.

```
vec3 color2 = color(r, world,0);
#pragma omp critical
col += color2;

}
```

Uma outra possibilidade de otimização seria utilizando tasks para fazer tarefas paralelas em situação que não dependem uma das outras. Essa solução não se apresentou muito eficaz, tendo em alguns casos apresentado desempenho ligeiramente pior que o caso sem task. Essa ligeira piora se deve ao custo de criar threads, que pode não fazer valer apenas e compensar a otimização:

```
#pragma omp parallel
{
    #pragma omp master
    {
        int ir,ig,ib;

        #pragma omp task shared(col)
        col = vec3( sqrt(col[0]), sqrt(col[1]), sqrt(col[2]) );

        #pragma omp taskwait
        #pragma omp task shared(ir)
        ir = int(255.99*col[0]);

        #pragma omp task shared (ig)
        ig = int(255.99*col[1]);

        #pragma omp task shared (ib)
        ib = int(255.99*col[2]);

        #pragma omp taskwait
        std::cout << ir << " " << ig << " " << ib << "\n";
    }

}
```

O uso de tasks devem ser feitas com cuidado, sempre considerando se realmente ela vai ou não ser eficiente no seu propósito de otimização.

3. Aplicação e Gráficos de Desempenho

Para otimizar o algoritmo, foram feitas algumas mudanças que impactaram significativamente no desempenho. O código fonte retirado de **https://github.com/petershirley/raytracinginoneweekend** inicialmente contava com uma imagem tamanho 1200x800, porém para que se pudesse realizar testes de desempenho em uma velocidade apropriada, diminuiu-se o tamanho da imagem em dez vezes. Além disso, adicionou-se uma variável de proporção chamada *'prop',* para caso deseje rodar o teste

para tamanhos maiores de imagem. Para fim dessa análise, usaremos o tamanho 144x96, tanto no código original quanto no otimizado.

a. Como repetir os testes

Para repetir os teste, basta clonar o github do link https://github.com/antoniosigrist/RayTracing. Localize o arquivo clonado em seu terminal e rode o comando **make.** Uma outra opção é compilá-lo na mão com os comandos **g++-6 main.cc -o main -fopenmp** seguido de **./main > imagem.ppm.** O código gerará um arquivo texto no seu diretório chamado **tempo.txt**, em que haverá dados de tempo e threads utilizados para cada teste. Um exemplo de saída de texto para o teste de 1 a 5 threads é:

Arquivo exemplo: tempo.txt

Número de Threads x Tempo de Execução:
N Threads: 1 - Tempo de Execução: 8.86228,
N Threads: 2 - Tempo de Execução: 5.80458,
N Threads: 3 - Tempo de Execução: 5.15364,
N Threads: 4 - Tempo de Execução: 5.24163,
N Threads: 5 - Tempo de Execução: 5.66867

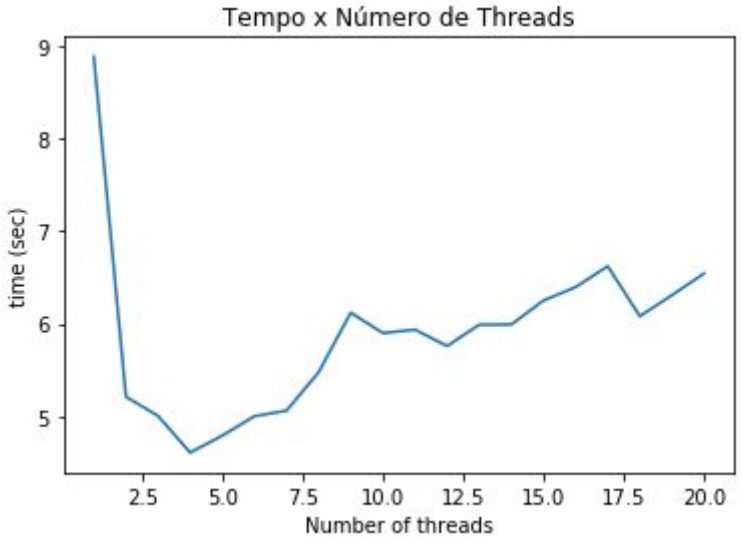
Para alterar o número de threads desejado para o teste, basta alterar a variável *tamanho_testes* no início da main. Caso não deseje que o teste inicie de uma unica thread, altere a variável *tamanho_testes_i.*

Para cada teste é aumentado gradualmente o número de threads e é retornado o tempo de execução de cada um deles. Para gerar os gráficos de desempenho, copie os dados gerados no arquivo tempo.txt para o ipython notebook chamado **Gráficos** onde estará indicado para fornecer os dados gerados.

b. Número de Threads e Tempo de Processamento

i. Resultados e Análise

O primeiro teste realizado foi para a imagem de 144x96, em que rodou-se o código inicialmente com uma thread, depois com duas até atingir vinte threads.

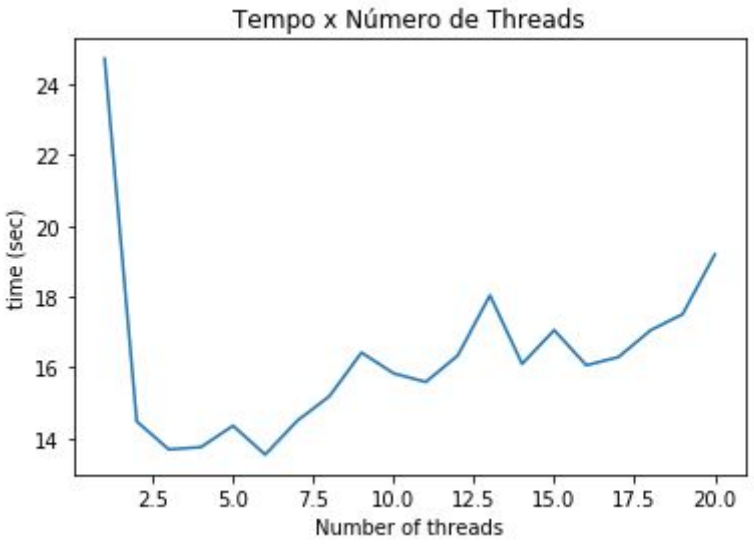


Nota-se que o primeiro ponto do gráfico, referente a uma única thread, teve 8,89 segundos de execução. Conforme começou-se a inserir mais threads, o tempo de execução foi diminuindo drasticamente até atingir seu menor valor de 4,60 segundos com quatro threads de execução, **48%** de otimização em relação a uma única thread que simboliza a não paralelização do código, o tempo do código original.

Conforme o número de threads começa a aumentar em relação a 4 threads, nota-se que o tempo de execução volta a aumentar de forma que podemos aproximar a um crescimento linear. Isso se deve ao custo de se criar threads.

Existe um trade-off entre criar threads para realizar tarefas, pois pode ser que o número de threads criadas demore mais que a otimização que elas conseguem gerar. Dessa forma, deve-se entender o problema para identificar qual o número ideal de threads para cada código, nesse caso quatro.

Deseja-se sabe se esse ganho de execução permanece constante para diferentes valores de tamanho da imagem e se existe algum tamanho de imagem que não seja otimizado com quatro threads rodando. Para uma imagem de tamanho 240x160, obteve-se um ganho de **45%.** Podemos ver que o ganho ainda é muito próximo e os dados podem ser obtidos na **Tabela II**. Nota-se, porém, que o melhor desempenho ocorreu com três threads ao invés de quatro como anteriormente.



Notamos que o formato da curva de desempenho conforme o número de threads aumenta é muito similar a curva gerada pela imagem 144x96. Isso nos é um bom indício para assumir que a inclinação da curva está diretamente ligada ao número de threads, assim como sua altura no mapa está relacionado com o tamanho da imagem.

Tamanho da Imagem	Ganho com Otimização
60x40	47%
120x80	44%
144x96	48%
240x160	45%

ii. Consumo de Memória

Para identificar o consumo de memória que o programa está usando, usou-se o monitor de atividade do Macbook para identificar o quanto estava sendo usado. Abaixo segue a tabela com o número de threads, tempo de execução e memória consumida durante o processamento para a imagem de 144x96:

Tabela I - Dados Imagem 144x96

Número Threads	Tempo de Processamento (segundos)	Consumo de Memória
1	8.88916	524KB
2	5.21028	580KB
3	5.00261	652KB
4	4.60633	708KB
5	4.7896	768KB
6	5.00054	816KB
7	5.06027	884KB
8	5.4754	936KB
9	6.12047	992KB
10	5.89879	996KB
11	5.93603	1,0MB
12	5.75827	1,1MB
13	5.98995	1,1MB
14	5.99168	1,2MB
15	6.25176	1,3MB
16	6.39559	1,3MB
17	6.62246	1,4MB
18	6.08146	1,4MB
19	6.30881	1,4MB
20	6.54377	1,5MB

Nota-se que por mais que o número de threads aumentar otimize o tempo (de 0 a 4 threads) o custo que se para por essa otimização é na memória de processamento necessária. Conforme aumenta o número de threads, aumento a memória necessária para processar os dados.

No entanto, desejamos entender se o uso de memória está diretamente ligado ao número de threads ou se também ao tamanho da imagem que estamos tratando. Para isso, rodamos o código novamente para o tamanho 240x160. Vemos que o consumo de memória continua muito similar ao caso anterior, atrelando, nesse caso, o grosso de processamento ao número de threads.

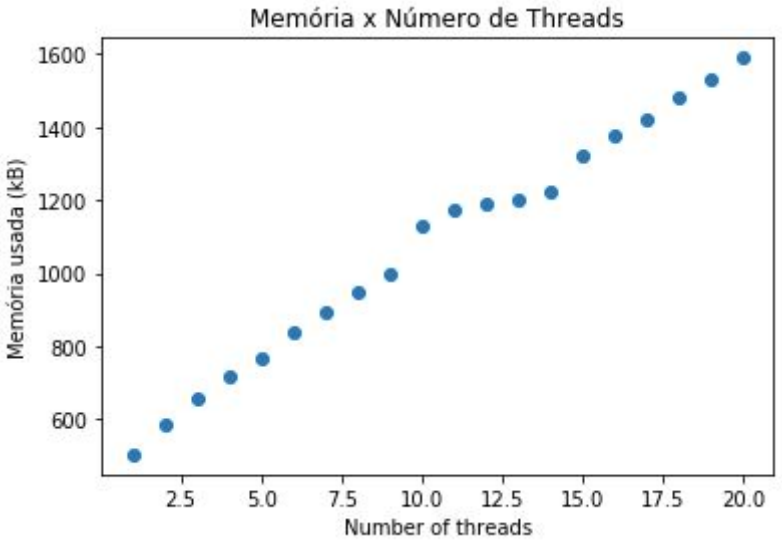
Tabela II - Dados Imagem 240x160

Número Threads	Tempo de Processamento (segundos)	Consumo de Memória
1	24,7289	504KB
2	14,4734	588KB
3	13,6823	656KB
4	13,7496	720KB
5	14,3514	768KB
6	13,5364	836KB
7	14,4948	892KB
8	15,1864	948KB
9	16,4197	1,0MB
10	1,5833	1,0MB
11	15,5894	1,1MB
12	16,3408	1,1MB
13	18,0387	1,2MB
14	16,1005	1,2MB
15	17,0539	1,3MB
16	16,0609	1,3MB
17	16,2932	1,4MB
18	17,05	1,4MB
19	17,5058	1,5MB
20	19,1951	1,5MB

4. Número de Threads x Consumo de Memória - Regressão

Podemos estimar qual será o consumo de memória regredindo os dados medidos.

Consumo de Memória = B0 + Número de Threads x B1 + Erro



A partir dos dados medidos, podemos estimar que para cada thread gerada, teremos um impacto de 55KB (B1) de memória utilizada. Esse resultado da regressão pode ser obtida no arquivo ipython notebook **Gráfico.**