Jean Luca de Oliveira Walper
Junho 2019

Super Computação

– Projeto 2 –
Ray tracing em CUDA
Link: https://github.com/Lightclawjl/RayTracingCuda
– Relatório –

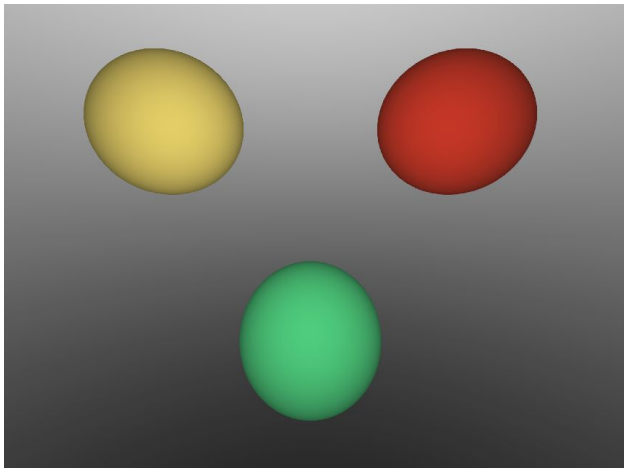
1.1 - Especificação do projeto

Esse projeto tem como objetivo criar um código paralelizado em GPU e analisar as suas vantagens. Para isso foi proposta a atividade de se criar um Ray Tracer em CUDA, e comparar o desempenho do código em GPU com uma versão sequencial do mesmo.

O Projeto 2 pode ser considerado uma continuação direta do Projeto 1, já que o objetivo é o mesmo (otimizar uma tarefa de Ray Tracing via paralelização de código), o que muda é a tecnologia utilizada para esse objetivo. (No primeiro projeto foram utilizadas Threads de CPU)

Foi usado como base um projeto de Ray Tracing aberto ao público, chamado “Ray Tracing in One Weekend”, feito por Peter Shirley. (disponivel em http://in1weekend.blogspot.com/2016/01/ray-tracing-in-one-weekend.html).

O script criado em CUDA para esse projeto equivale ao código resultante do *Capítulo 06 - Antialiasing*. Diferentemente do projeto 1, onde o projeto completo foi modificado para ser paralelizado, agora no Projeto 2 o código em GPU foi escrito do zero e paralelizado em sessões. O código avançado o bastante para ter um sistema de antialising é o bastante para o escopo do projeto.



Por conta disso existem algumas diferenças em escolhas de design do algoritmo do Shirley e do criado para esse projeto, mas essas diferenças são pouco relevantes em tempo de execução (e quase sempre fazem parte do overhead do código).

1.2 - Funcionamento de um Ray Tracer

Ray Tracing é uma técnica de renderização gráfica 3D, baseada em simular os efeitos de iluminação em várias superfícies com materiais diversos. Para isso, um Ray Tracer simula o comportamento de raios de luz individuais.

Esses raios são simulados reversamente, já que partem da câmera em direção a cena, com as informações das superfícies que esse raio atinge, é possível calcular qual seria a cor do raio de luz que fizesse esse mesmo caminho, mas em direção a câmera.

Para cada pixel, um ou mais raios devem ser lançados, assim dizendo para o programa qual a cor de dado pixel. Essa técnica permite cálculos de refração, reflexão e difusão de raios de luz, gerando imagens super realistas e complexas.

1.3 - Paralelização em CUDA

Normalmente códigos e scripts da área da computação são planejados para rodar em CPU, mas em alguns casos é possível realizar essas tarefas rodando scripts arquitetados para GPU. Scripts de GPU sao melhores para resolver problemas onde há tarefas simples que podem ser realizadas simultaneamente.

A principal diferença da GPU pra CPU é que enquanto uma CPU tem poucos núcleos de processamento capazes de realizar várias tarefas e que operam com clocks bem mais altos, a GPU tem milhares de núcleos, mas cada um bem mais simples e otimizado para operações de ponto flutuante (float).

Essa quantidade de núcleos da GPU permite que ela realize seu propósito mais básico, e o que deu o nome dessa arquitetura, renderizar imagens. Porém ultimamente cada vez mais as GPUs são usadas nas mais diversas tarefas, desde machine learning, big data, mineração de criptomoedas e etc…

CUDA é uma tecnologia proprietária da NVidia, com ela é possível compilar códigos de C++ para serem rodados nas arquiteturas das GPUs GeForce.

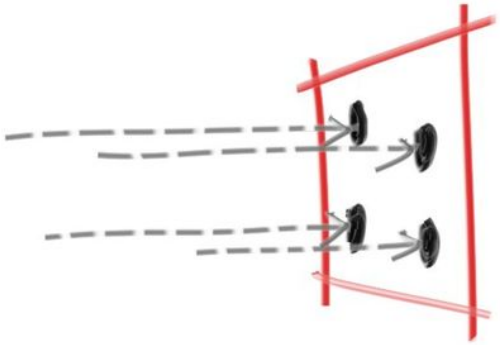
Geralmente não é possível paralelizar um script/projeto inteiro que rode em CPU para GPU, mas é quase sempre provável que haja sessões do código paralelizados. Então um código em CUDA costumam ter uma parte de *Overhead* que ainda é executada na CPU.

O seguinte post no fórum de CUDA foi usado como um bom “Starting point” para o projeto: https://devblogs.nvidia.com/even-easier-introduction-cuda/

1.4 - Anti aliasing

Lançar um raio por pixel já é o bastante para que seja possível renderizar uma imagem condizente com a cena desejada, mas será possível perceber mudanças brutas na cor de pixels vizinhos em uma imagem gerada assim, nos contornos dos objetos. Essa mudança brusca dá a sensação de que a imagem tem “serrilhados”.

Para gerar uma imagem sem “serrilhados” nas bordas dos objetos, para cada pixel são lançados um número de raios em direções levemente diferentes, mas que ainda pertencem à mesma área da imagem na qual o pixel representa.



Com isso, a cor do pixel é definida pela média desses raios, fazendo com que pixels nas bordas dos objetos tenha um efeito de transição mais suavizado de cor com o que está por traz.

Essa técnica é conhecida como Anti aliasing.

1.4.1 - Antialiasing Pseudo Aleatorio

Uma das dificuldades do Projeto era calcular os pontos internos do pixel aleatoriamente para realizar o Antialiasing, Para contornar esse problema foi criado uma função Pseudo-Aleatória dentro do código, que podia ser chamada pela GPU.

A função é considerada Pseudo-Aleatória por que apesar de ser caótica, o mesmo input ainda irá retornar sempre o mesmo valor.

```
__device__
//return a "random" number betwen 1 and -1 based on 4 floats as seed (it is not random actually, but it is enough)
float pseudo_rand(float a, float b, float c, float d){

    //a controlled chaotic expression
    float val = ( (((a + 134.854f + c)/3.3f) * c) / (d + 3.645f) );
    if(val == 0.0f){
        //if a or c is zero, use this other expression
        val = (c + d + 89.423f) * 9.308f * d * 1.54f + c;
    }
    val *= 11.245f;

    //val = val % 2; //I cant use modulo on floats inside CUDA!!!
    //workaround:
    int precision = 100000; //how many decimal slots i want to keep (log(10), 5 in this case)
    int ret = (int) val % (2 * precision); //module it with some precision
    val = (float)ret / (precision); // make ret a floating point

    return (val - 1.0f);
}
```

Nessa função 4 floats são usados como seed. Esse floats são parâmetros como o Id do pixel, a resolução e etc…

A implicação disso é que o antialiasing vai ser constante para cada posição na tela. Também para melhorar o efeito e deixá lo menos constante, após cada raio lançado, o cálculo de quanto movê-lo para o próximo raio do mesmo pixel, o que faz com que alguns desses raios consigam ser lançados fora do escopo de 1 pixel, então eles podem invadir a área de outro pixel e usar essa informação para o cálculo da sua própria cor.

Isso não é um problema de nenhuma forma para a comparação de desempenho entre os códigos.

1.5 - Estrategias

Paralelização em GPU permite-se criar um número gigantesco de Threads e fazer com que cada uma realize a rotina especificada. Mudando alguns parâmetros dessa rotina (geralmente o Id da thread) é possível criar um código que realiza a mesma tarefa para cada pedaço de um input bem maior.

No caso deste projeto, é criada uma thread para cada pixel, e cada uma realiza o cálculo do Ray Tracing apenas nesse pixel, todas simultaneamente, e retornam os valores de cores de cada pixel para um vetor salvo em uma área de memória compartilhada entre a CPU e a GPU. (essa area de memoria compartilhada se chama Unified Memory: ver https://devblogs.nvidia.com/unified-memory-in-cuda-6/).

Após o calculo das cores, o código iterativamente escreve um arquivo de imagem com as informações do vetor. Essa etapa do código é de longe a mais demorada após a paralelização.

2.1 - Ambiente de teste

- A paralelização foi feita com CUDA e compilada com nvcc V10.0.130 e gcc 7.3.1
- Foi comparada com a versão do capitulo 6 do projeto de referencia, encontrado neste repositorio: https://github.com/pfranz/raytracinginoneweekend/tree/ch06_antialiasing.
- As imagens foram geradas em resoluções 4K (3840 x 2160 pixels).
- Os testes foram rodados em um sistema Linux 64 bits, utilizando um processador Intel Core i7-8750H, de 2.20 GHZ e 12 núcleos e uma Geforce GTX 1070 Max-Q (Versão de laptop).
- Cada teste foi rodado ao menos 5 vezes e tirada a média dos valores.

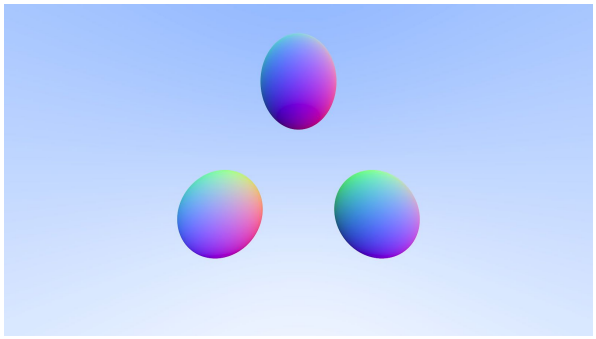
- Os valores de tempo foram calculados utilizando a biblioteca Chronos do C++.
- Todos os testes geraram uma imagem equivalente.

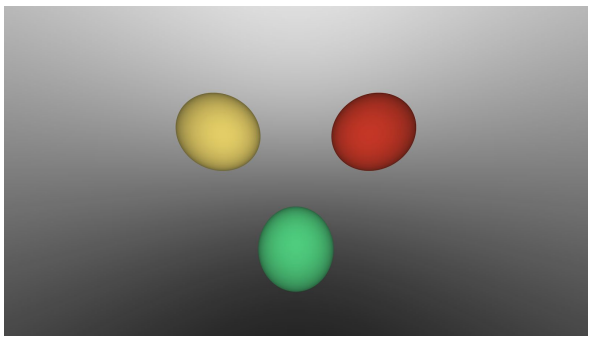
2.2 Analise de desempenho

Vou analisar o tempo de execução de 2 sessões do código, a sessão de loop, onde o cálculo do Ray tracer acontece, e a sessão de escrita do arquivo na memória do computador.

As imagens são levemente diferentes apenas na cor e alguns cálculos de Field Of View (FOV). mas essas poucas mudanças não influenciam as análises.

Abaixo uma comparação, a gerada pela CPU e então a gerada pela GPU.





O output que criado para testes devolve em segundos o tempo de execução das sessões.

```
Starting CPU loop section.
Writing file.
Done.
CPU   time taken: 19.178494
Write time taken: 1.168468
Total time taken: 20.346962
```

```
Starting GPU Parallelized section.
Writing File
GPU   time taken: 0.028357
Write time taken: 1.312299
Total time taken: 1.340656
```

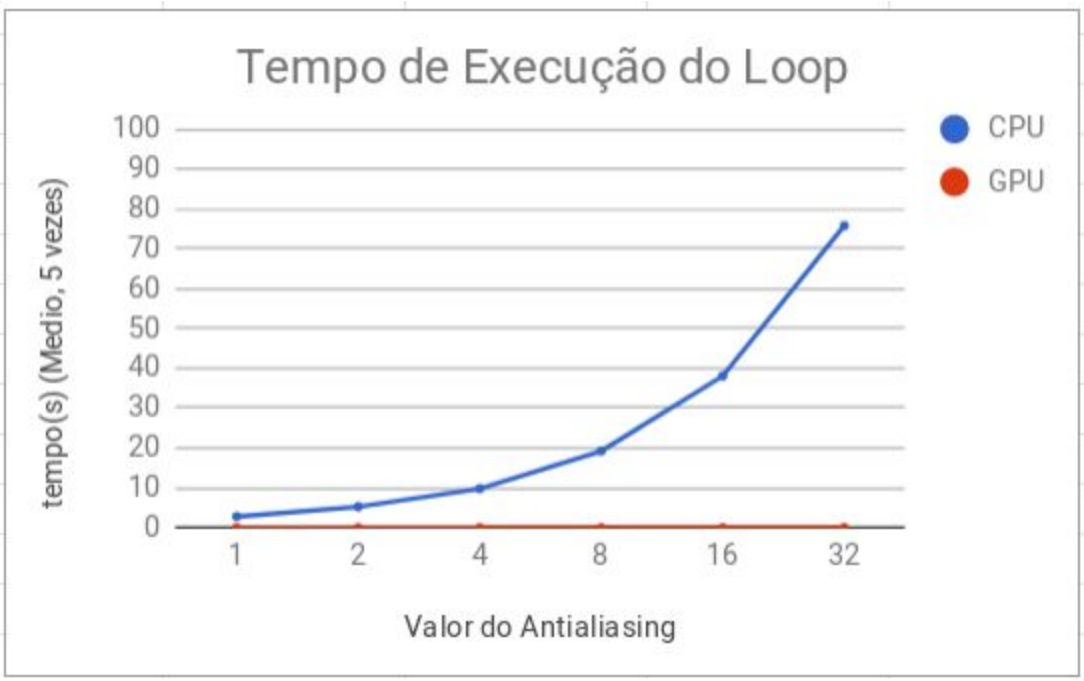
Com apenas esse caso exemplo já é possível ver a diferença entre a velocidade dos códigos.

O antialiasing é importante para essa análise, já que um valor maior de antialiasing significa mais raios sendo lançados para cada pixel, essencialmente multiplicando o payload do código.

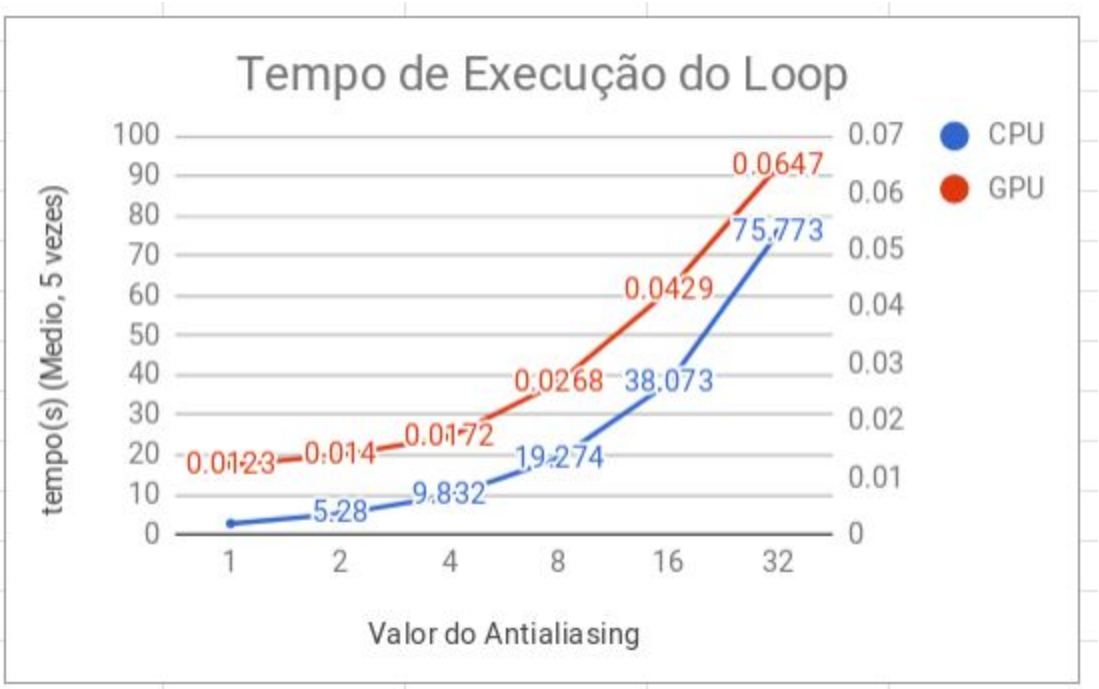
Abaixo uma tabela de tempo de processamento (em segundos) da parte do loop do projeto:

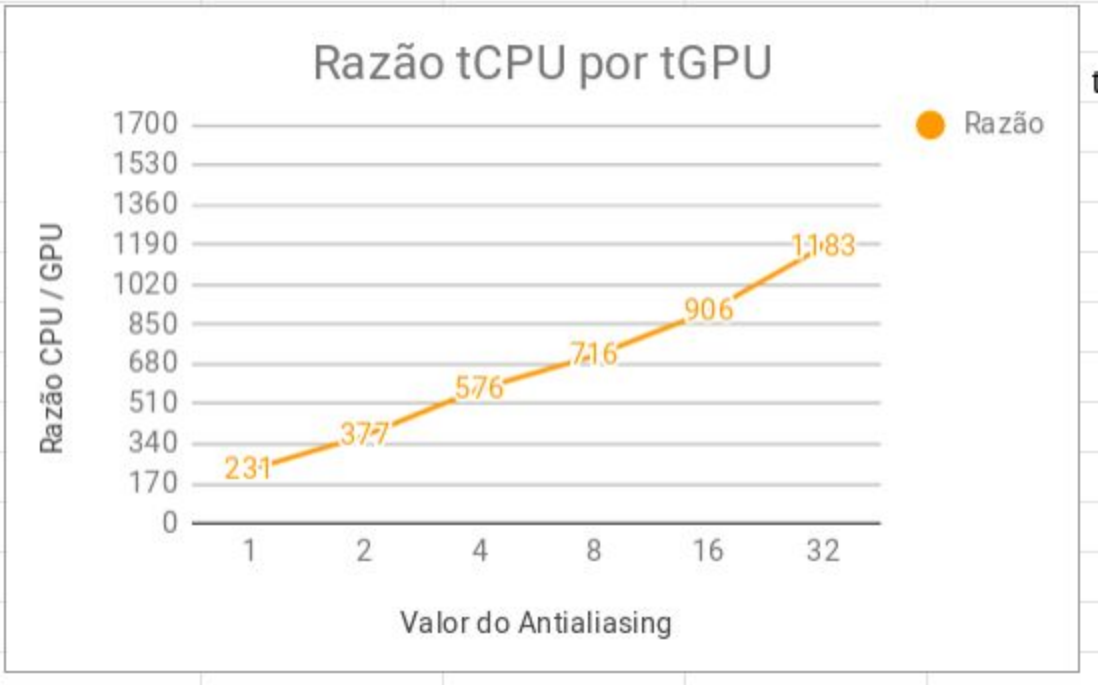
Numero de AA	CPU(s)	GPU(s)
1	2.85	0.0123
2	5.28	0.014
4	9.832	0.0172
8	19.274	0.0268
16	38.073	0.0429
32	75.773	0.0647

Já é fácil perceber a diferença absurda entre o código iterativa e o paralelo. Abaixo um gráfico comparando os dois valores para colocar em perspectiva:



A diferença é tão grande que para analisar melhor, vamos jogar os valores do tempo da execução em GPU para o eixo da direita, também vamos dar uma olhada em um gráfico da razão entre o tempo de execução da GPU dividida pelo tempo de execução da CPU:





Ficou fácil perceber a vantagem de converter esse projeto para GPU. O poder de processamento de poucos núcleos da CPU é bem maior, mas é impossível ele ganhar contra milhares de pequenos processadores da GPU em casos como esse. Mas a diferença de desempenho não é a única coisa interessante aqui.

Podemos ver que a progressão do tempo em ambos os casos é bem parecida. A perda em tempo de processamento (ou seja, demorar mais para processar) com mais níveis de AA é o esperado, com o dobro de níveis de AA, o dobro de tempo de processamento.

Mas quando olhamos a razão entre os tempos, vemos que a progressão dela é logarítmica. Mostrando que a paralelização pode ser muito mais beneficial em casos específicos, geralmente com quantidades maiores de dados ou de operações.

O porquê desse comportamento provavelmente se dá por conta do acesso a memória. No código em GPU, uma thread recebe toda informação necessária e roda todos os níveis do AA de uma única vez, sem necessitar acessar a memória compartilhada (unified memory) novamente.

A transferência de memória do ambiente da CPU para o da GPU custa caro, e mesmo usando a Unified Memory, temos um intermediário de memória, temos aqui evidências de que esse tipo de alocamento não será melhor do que memória alocada apenas para a CPU e memória alocada apenas para a GPU.

Podemos ver isso evidenciado também no tempo de escrita do arquivo de imagens:

```
Done.
CPU   time taken: 19.270837
Write time taken: 1.175680
Total time taken: 20.446517
```

```
GPU   time taken: 0.063740
Write time taken: 1.340998
Total time taken: 1.404739
DONE
```

O tempo de escrita é bem constante entre os testes (sempre na mesma resolução, e ambos usam o mesmo método de stream). E a escrita da versão de CPU é consistentemente melhor do que a da versão de GPU.

Isso acontece porque nessa parte do código, a CPU está acessando um vector salvo na memória ram do computador, fazendo uso de runtime cache e outras facilidades para otimizar esse tempo. Já a versão de GPU tira essa informação do espaço de Unified Memory, por isso acaba demorando um pouco mais. Lembrando que nessa sessão do código, em ambos os casos, o código está sendo rodado em um loop na CPU.

3 Conclusão

Ficou fácil perceber as vantagens de paralelizar tarefas de um código em GPU quando possível. Também foi fácil perceber que esse tipo de paralelização é muito impactante em grandes quantidades de tarefas simples ou dados que precisam ser analisados ou processados por tarefas pequenas.

Com a análise da razão entre os tempos e do tempo de escrita dos arquivos, foi possível mostrar a questão do compartilhamento de memória entre CPU e GPU. Foi utilizado a tecnologia de Unified Memory, com o intuito de facilitar o desenvolvimento do projeto, mas acredito que seria possível otimizar mais ainda o tempo do loop em GPU utilizando técnicas mais convencionais de alocamento e compartilhamento de memória.

Como ficou claro a vantagem do uso da GPU, fiz um teste com 128 níveis de Anti Aliasing.

```
GPU   time taken: 0.229287
Write time taken: 1.359562
Total time taken: 1.588849
```

```
CPU   time taken: 304.123431
Write time taken: 1.186955
Total time taken: 305.310386
```

Pessoalmente preciso falar que ainda acho a diferença impressionante, e espero trabalhar de novo com codigo em GPU.