UNIVERSIDADE FEDERAL DE SÃO CARLOS
Centro de Ciências Exatas e de Tecnologia
Departamento de Computação

Arquiteturas de Alto Desempenho - Semana 5

Conjunto de Mandelbrot

Gerando Fractais com o uso de GPUs

Professor: Dr. Emerson Carlos Pedrino

Víctor Cora Colombo. RA 727356. Engenharia de Computação.

São Carlos, 14 de Dezembro de 2020

Sumário

1 Progamando em GPUs com MATLAB

Quando programamos normalmente, costumamos escrever códigos voltados para serem executados de forma sequencial, sem qualquer paralelismo. Esse tipo de programação é mais simples, e é o que é ensinado em quase todos os cursos de programação básica.

Vamos começar com um exemplo simples para calcular a Transformada de Fourier no MATLAB:

Código 1 – Transformada de Fourier sequencial

```
>> A1 = rand(5000, 5000);
>> tic;
>> B1 = fft(A1);
>> toc;
Elapsed time is 0.067589 seconds.
```

Nesse trecho de código, criamos uma matriz quadrada de ordem $N = 5000$, e aplicamos a Transformada de Fourier sobre ela. O programa demorou cerca 0.1 segundos para executar. Parece rápido o suficiente para um usuário comum. Porém, e se estivermos mexendo com aplicações em níveis industriais, com valores muito maiores que 5000? Ou talvez seja necessário executar a Transformada para milhares de matrizes diferentes em um curto período de tempo. Da forma como está no momento, o código é executado sequencialmente, em uma única CPU, de forma que o tempo de execução escala linearmente de acordo com o número de instruções.

As GPUs podem ser utilizadas para paralelizar cálculos matemáticos em larga escala, diminuindo seu tempo de execução e permitir alta escalabilidade. Para utilizar a GPU da sua máquina no MATLAB, primeiramente vamos verificar se ela está sendo reconhecida:

Código 2 – Verificando a GPU

```
>> gpuDevice
ans =

  CUDADevice with properties:

                      Name: 'GeForce GTX 770'
                     Index: 1
         ComputeCapability: '3.0'
            SupportsDouble: 1
             DriverVersion: 11.1000
            ToolkitVersion: 10.2000
        MaxThreadsPerBlock: 1024
```

```
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 2.0938e+09
           AvailableMemory: 1.5671e+09
       MultiprocessorCount: 8
              ClockRateKHz: 1202000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
          CanMapHostMemory: 1
           DeviceSupported: 1
            DeviceSelected: 1
```

Parece tudo certo. A minha placa de vídeo decidada está sendo corretamente mostrada. Se não for o caso na sua máquina, pode ser necessário atualizar os drivers de vídeo da *NVIDIA*.

Então, para transforma o código sequencial em paralelo e executá-lo na GPU, podemos usar a função `gpuArray()`, que copia uma matriz (ou vetor) para a GPU, e garante que qualquer cálculo sobre esses dados seja feito dentro dela:

Código 3 – Transformada de Fourier na GPU

```
>> A2 = gpuArray(A1);
>> tic;
>> B2 = fft(A2);
>> toc;
Elapsed time is 0.002190 seconds.
```

Adicionando uma simples função, conseguimos um *speedup* de quase 27 vezes mais rápido que o tempo de execução do código sequencial! Não é preciso nenhum conhecimento profundo ou código super complexo para conseguir rodar códigos de forma paralela dentro da GPU do seu computador pessoal.

Contudo, há um problema que não parece óbvio a princípio: e se levarmos em conta também o tempo para copiar a matriz para a GPU?

Código 4 – Tempo para copiar a matriz para a GPU

```
>> tic;
>> A2 = gpuArray(A1);
```

```
>> B2 = fft(A2);
>> toc;
Elapsed time is 0.091493 seconds.
```

Agora o tempo de execução é pior do que antes! Se mover a matriz para a GPU é tão lento, parece não valer a pena executar o código na GPU. Porém, há um detalhe nesse código que nos permitirá solucionar esse problema. A matriz `A1` não depende de nenhum cálculo prévio. Ela é criada de forma aleatória sempre que o código é executado.

E se, ao invés de copiar a matriz inteira, já criarmos a matriz aleatória diretamente dentro da GPU?

Código 5 – Criando a matriz direto na GPU

```
>> tic;
>> A3 = rand(size(A1), 'gpuArray');
>> B3 = fft(A3);
>> toc;
Elapsed time is 0.000980 seconds.
```

Com essa solução, o *speedup* não só foi recuperado, como foi aumentado para quase **70 vezes**.

2 Criando fractais com o Conjunto de Mandelbrot

Passada a introdução à programação paralela em MATLAB com o uso de GPUs, vamos agora ver como criar fractais por meio da técnica do Conjunto de Mandelbrot[1].

A ideia básica aqui é utilizar cálculos no conjunto dos números complexos para gerar as formas do fractal. Define-se um plano 2D limitado por $xlim$ no eixo x e $ylim$ no eixo y. Será mostrado então o fractal com zoom nesse trecho do resultado. Para simplificar, nesse trabalho será mostrado o fractal inteiro, ou seja, $xlim = [-2.5, 1.0]$ e $ylim = [-1.25, 1.25]$.

O método também precisa que seja definida a quantidade máxima de iterações N. Quanto maior o número de iterações, mais "desenhado"ficará o fractal, de forma que seria possível aumentar muito o zoom e continuar vendo formas significativas. Para efeitos de demonstração, será usado $N = 1000$, com o intuito de visualizar a performance do programa.

É criado uma matriz $z0$ de tamanho $width$ por $height$, em cada cada entrada da matriz é um número complexo, com parte real dentro do limite $xlim$, e parte complexa dentro do

[1] https://www.math.univ-toulouse.fr/~cheritat/wiki-draw/index.php/Mandelbrot_set

limite $ylim$. Por fim, inicializa-se uma matriz $count$ de mesmo tamanho, preenchida com zeros. No fim, essa matriz representará a imagem gerada.

O algoritmo em si é simples: Inicializa-se $z = z0$, faz-se uma operação *element-wise* sobre z e verifica-se se o módulo de cada entrada resultante é menor ou igual a 2. Se sim, adiciona-se 1 na posição correspondente em $count$. Repete-se esse processo N vezes, e no final o resultado de $count$ é a matriz que representa a imagem do fractal.

Código 6 – mandelbrot.m

```
function count = mandelbrot(z0, count, N)
    z = z0;
    for n = 0:N
        z = z.*z + z0;
        inside = abs(z) <= 2;
        count = count + inside;
    end
end
```

2.1 Mandelbrot sequencial

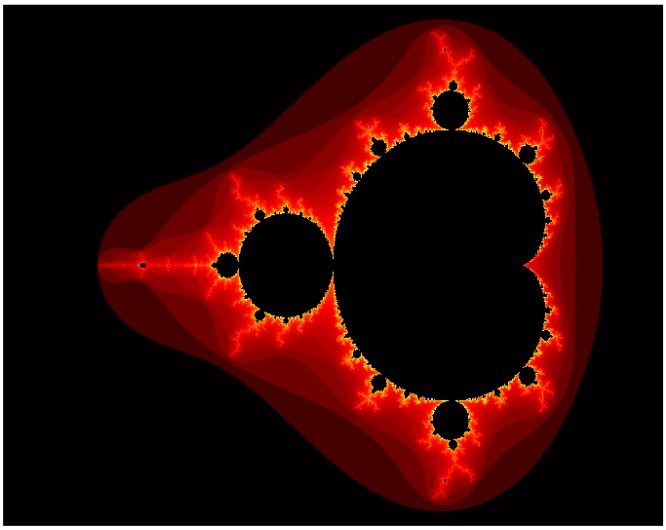
Com a função para encontrar o fractal já definida, basta chamá-la passando os parâmetros necessários:

Código 7 – Execução sequencial (CPU)

```
N = 1000; % numero maximo de iteracoes
width = 1000; % largura da imagem resultante
height = 1000; % altura da imagem resultante
xlim = [-2.5, 1.0];
ylim = [-1.25, 1.25];

x = linspace(xlim(1),xlim(2), width);
y = linspace(ylim(1),ylim(2), height);
[xGrid,yGrid] = meshgrid(x,y);
z0 = complex(xGrid,yGrid);
count = zeros(size(z0));

tic;
count = mandelbrot(z0, count, N);
toc;

```

```
17 % Mostrar a imagem
18 imagesc(x, y, log(count));
19 colormap([hot(); 0 0 0; 0 0 0]);
20 axis off;
21
22 % Elapsed time is 5.364477 seconds.
```

Figura 1 – Fractal resultante.



Fonte: Feito pela autor no MATLAB.

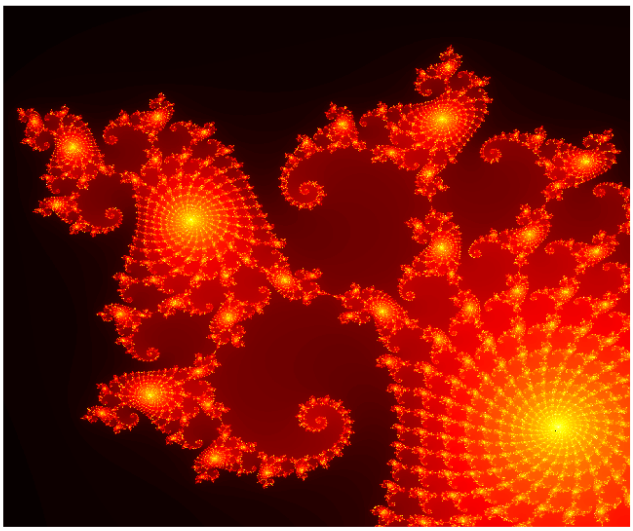
Temos então nosso fractal! Para conseguir imagens diferentes, há a possibilidade de mexer tanto na equação da linha 4 do Código 6, quanto nos limites. Por exemplo, fazendo $xlim = [-0.748766713922161, -0.748766707771757]$ e $ylim = [0.123640844894862, 0.123640851045266]$ temos o fractal da Figura 2.

2.2 Mandelbrot na GPU

Agora, e se quisermos aumentar a resolução da imagem, ou o valor de N, por exemplo? Do jeito que está agora, já está demorando 5 segundos. Um pequeno aumento nos parâmetros pode facilmente levar a execução para a casa dos minutos.

Podemos então usar as técnicas de paralelismo com GPU que vimos anteriormente para diminuir o tempo de execução e tornar o programa escalável. Para isso, precisamos identificar os pontos que precisam ser modificados no Código 7. A ideia é trocar as matrizes e vetores "normais"pelas alternativas de GPU que o MATLAB provém.

Figura 2 – Outro fractal.



Fonte: Feito pela autor no MATLAB.

As matrizes recebidas pela função mandelbrot() são $z0$ e $count$. Para a primeira, trocaremos `linspace()` por `gpuArray.linspace()`. É possível verificar que isso realmente transforma $z0$ em uma `gpuArray` chamando `class(z0)`.

Para $count$, a função `zeros()` será mudada para `zeros(size(z0), 'gpuArray')`. Isso faz com que a criação da matriz aconteça diretamente dentro da GPU, economizando o tempo que seria gasto com a cópia desses dados.

Código 8 – Mandelbrot na GPU

```
x = gpuArray.linspace(xlim(1),xlim(2), width);
y = gpuArray.linspace(ylim(1),ylim(2), height);
[xGrid,yGrid] = meshgrid(x,y);
z0 = complex(xGrid,yGrid);
count = zeros(size(z0), 'gpuArray');

tic;
count = mandelbrot(z0, count, N);
toc;

% Elapsed time is 0.059046 seconds.
```

Temos dessa vez então um *speedup* de 90 vezes. Isso também permite que modifiquemos os parâmetros com um impacto menor no tempo de execução. O único limitante é a memória da GPU: enquanto a CPU possui acesso à uma RAM de 16GB na minha máquina,

a GPU possui apenas 2GB de VRAM. Isso limita o tamanho das matrizes que podem ser alocadas dentro dela.

2.3 Melhorando ainda mais o speedup: arrayfun

E se eu dissesse que é possível melhorar ainda mais o tempo de execução? Para isso, será usada a função `arrayfun()`. Essa função melhora a performance de códigos paralelizáveis, como foi visto em um trabalho anterior. A sintaxe pode ser vista no código a seguir:

Código 9 – Mandelbrot na GPU com arrayfun

```
x = gpuArray.linspace(xlim(1),xlim(2), width);
y = gpuArray.linspace(ylim(1),ylim(2), height);
[xGrid,yGrid] = meshgrid(x,y);
z0 = complex(xGrid,yGrid);
count = zeros(size(z0), 'gpuArray');

tic;
count = arrayfun(@mandelbrot, z0, count, N);
toc;

% Elapsed time is 0.000276 seconds.
```

Agora o *speedup* calculado foi de quase **20 mil vezes** em relação à execução sequencial.

3 Concluindo

Neste trabalho foi mostrado como melhorar a performance de códigos em MATLAB de forma significativa usando paralelismo na GPU. As modificações para isso são mínimas, de forma que não é difícil implementá-las.

Com esse conhecimento, foi mostrado como aplicar essas melhorias no cenário do cálculo da Transformada de Fourier e na geração de Fractais pelo método de Mandelbrot. Nesse último, foi possível alcançar um *speedup* na casa das dezenas de milhar.