

UNIVERSIDADE DE BRASÍLIA  
Faculdade do Gama

Programação para Sistemas Paralelos e Distribuídos

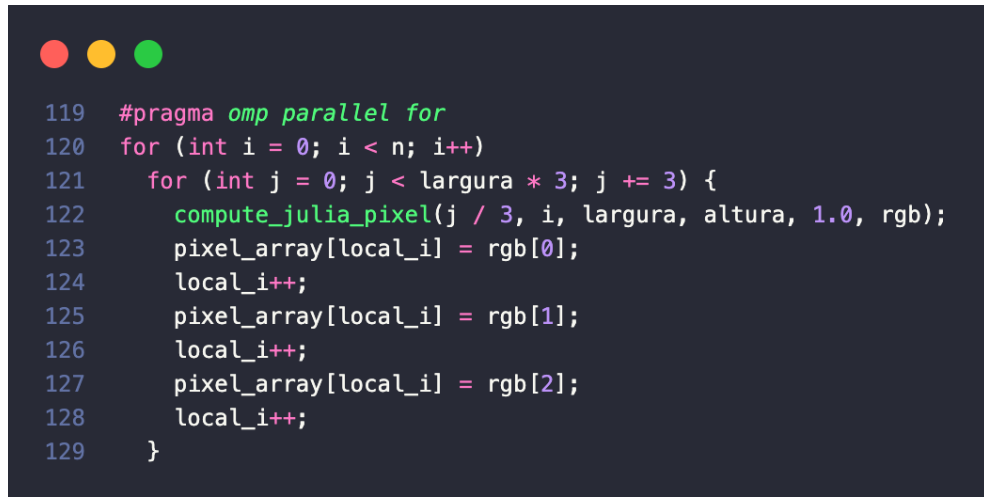
**Laboratório sobre programação OMP, CUDA e OpenCL**

Antônio Aldisio 20/2028211  
Fernando Miranda Calil 19/0106566  
Lorrany Oliveira Souza 18/0113992

Brasília, DF  
2023

## 1. OMP

A figura 01 é referente a parte que foi paralelizada. Como o `#pragma omp parallel for` não precisa ser fechada explicitamente, pois ele é uma diretiva do OpenMP que informa ao compilador como paralelizar um loop, ou seja, assim que o loop terminar, a paralelização será encerrada automaticamente.

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in C++ and uses OpenMP for parallelization. It consists of a nested loop: an outer loop for 'i' from 0 to n, and an inner loop for 'j' from 0 to 'largura \* 3' in increments of 3. The inner loop calls 'compute\_julia\_pixel' and updates 'pixel\_array' at 'local\_i' for the red, green, and blue channels. The OpenMP directive '#pragma omp parallel for' is placed at the start of the outer loop.

```
119 #pragma omp parallel for
120 for (int i = 0; i < n; i++)
121     for (int j = 0; j < largura * 3; j += 3) {
122         compute_julia_pixel(j / 3, i, largura, altura, 1.0, rgb);
123         pixel_array[local_i] = rgb[0];
124         local_i++;
125         pixel_array[local_i] = rgb[1];
126         local_i++;
127         pixel_array[local_i] = rgb[2];
128         local_i++;
129     }
```

Figura 01 - Trecho do código OpenMP

### 1.1 Dificuldades

O desenvolvimento do código foi realizado tranquilamente, pois foi necessário apenas inserir o `#pragma omp parallel for`.

## 1.2 Resultado

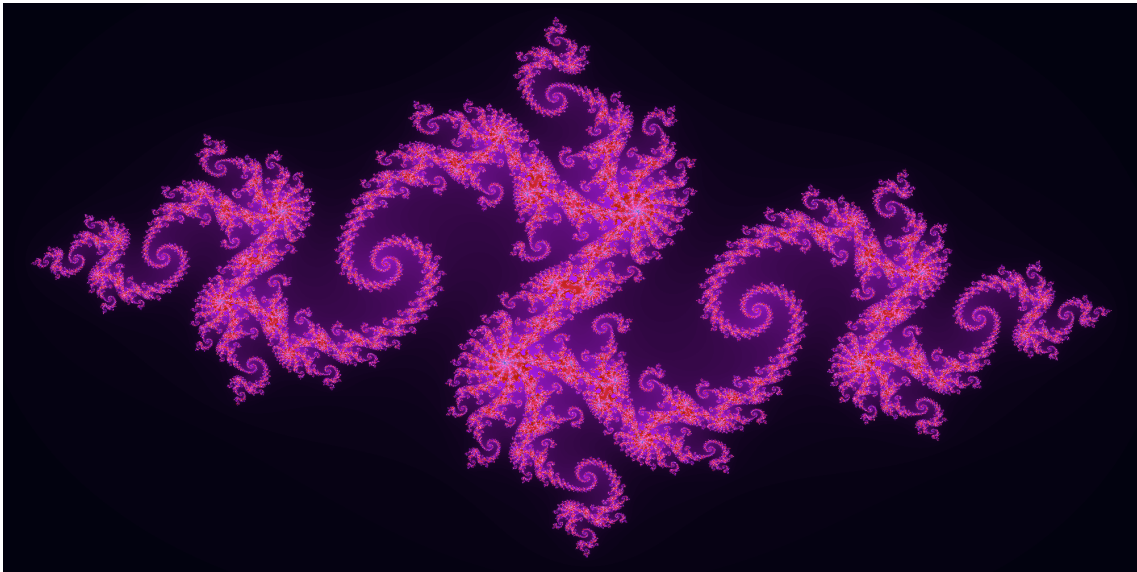


Figura 02 - Fractal do código OpenMP

## 2. CUDA

A figura 03 é referente à parte chamada da parte paralelizada. Isso inicia a execução do kernel na GPU, indicando o número de blocos e threads por bloco a serem utilizados. O kernel será executado em paralelo por todas as threads da GPU.

```
143 compute_julia_pixels<<<numBlocks, threadsPerBlock>>>(d_pixel_array, largura, altura);
```

Figura 03 - Trecho do código CUDA

Como visto na figura 04, temos a estrutura `dim3` é usada para representar as dimensões de blocos e grades de threads em uma execução paralela na GPU. O `numBlocks` é calculado com base nas dimensões do

problema, como a largura e a altura. É utilizado um arredondamento para cima para garantir que não haja threads sobrando sem trabalho. Esse cálculo determina a quantidade de blocos necessários em cada dimensão para cobrir todos os elementos do problema. Por outro lado, `threadsPerBlock` é uma estrutura especial usada para especificar as dimensões dos blocos de threads. No exemplo mencionado, `threadsPerBlock` é definido como um bloco 2D com 2 threads em cada dimensão, totalizando 4 threads por bloco.

```
131 dim3 threadsPerBlock(2, 2);
132 dim3 numBlocks((largura + threadsPerBlock.x - 1) / threadsPerBlock.x, (altura + threadsPerBlock.y - 1) / threadsPerBlock.y);
133
```

Figura 04 - Trecho do código onde tem a estrutura dim3

Na figura 05 temos a função `__global__` que será invocada a partir da CPU e será executada pela GPU e dentro dessa função temos `compute_julia_pixel` que é um função `__device__`, ou seja, é usada para declarar funções que serão executadas exclusivamente pela GPU que pode ser visto na figura 06.

```
55 __global__ void compute_julia_pixels(unsigned char *pixel_array, int largura, int altura) {
56     int x = blockIdx.x * blockDim.x + threadIdx.x;
57     int y = blockIdx.y * blockDim.y + threadIdx.y;
58     int area = largura * altura;
59     if (x < largura && y < altura) {
60         int local_i = (y * largura + x) * 3;
61         unsigned char rgb[3];
62         compute_julia_pixel(x, y, largura, altura, 1.0, rgb);
63         pixel_array[local_i] = rgb[0];
64         pixel_array[local_i + 1] = rgb[1];
65         pixel_array[local_i + 2] = rgb[2];
66     }
67 }
```

Figura 05 - Trecho do código que é paralelizado.

```

21 __device__ int compute_julia_pixel(int x, int y, int largura, int altura, float tint_bias, unsigned char *rgb) {
22     // Check coordinates
23     if ((x < 0) || (x >= largura) || (y < 0) || (y >= altura)) {
24         printf("Invalid (%d,%d) pixel coordinates in a %d x %d image\n", x, y, largura, altura);
25         return -1;
26     }
27     // "Zoom in" to a pleasing view of the Julia set
28     float X_MIN = -1.6, X_MAX = 1.6, Y_MIN = -0.9, Y_MAX = +0.9;
29     float float_y = (Y_MAX - Y_MIN) * (float)y / altura + Y_MIN ;
30     float float_x = (X_MAX - X_MIN) * (float)x / largura + X_MIN ;
31     // Point that defines the Julia set
32     float julia_real = -.79;
33     float julia_img = .15;
34     // Maximum number of iteration
35     int max_iter = 300;
36     // Compute the complex series convergence
37     float real=float_y, img=float_x;
38     int num_iter = max_iter;
39     while ((img * img + real * real < 2 * 2) && ( num_iter > 0 )) {
40         float xtemp = img * img - real * real + julia_real;
41         real = 2 * img * real + julia_img;
42         img = xtemp;
43         num_iter--;
44     }
45
46     // Paint pixel based on how many iterations were used, using some funky colors
47     float color_bias = (float) num_iter / max_iter;
48     rgb[0] = (num_iter == 0 ? 200 : - 500.0 * pow(tint_bias, 1.2) * pow(color_bias, 1.6));
49     rgb[1] = (num_iter == 0 ? 100 : -255.0 * pow(color_bias, 0.3));
50     rgb[2] = (num_iter == 0 ? 100 : 255 - 255.0 * pow(tint_bias, 1.2) * pow(color_bias, 3.0));
51
52     return 0;
53 }

```

Figura 06 - Trecho do código CUDA com a função compute\_julia\_pixel

## 2.2 Dificuldades

No desenvolvimento desse código foi passado por diversas dificuldades

- Definir e configurar de forma adequada o numBlocks e threadsPerBlock
- Alocação de memória para construção do fractal
- Cor do Fractal

## 2.3 Resultados

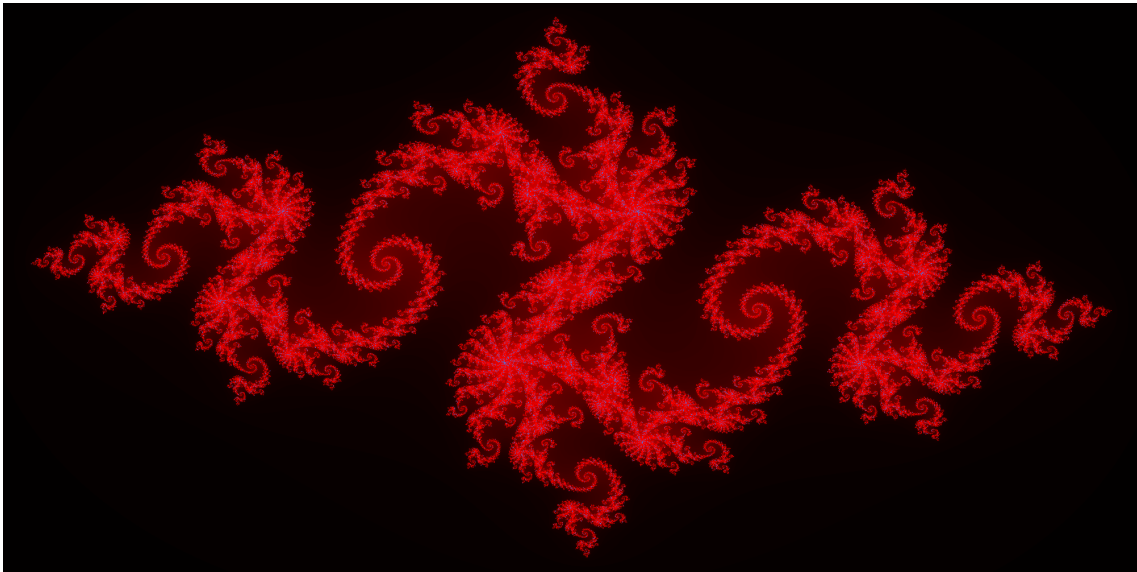


Figura 07 - Fractal do código CUDA

## 3. OpenCL

Na figura 08 temos a chamada que paraleliza a execução do fractal.

```
184 ret = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL, global_item_size, local_item_size, 0, NULL, NULL);  
185
```

Figura 08 - Trecho do código onde é chamado o código OpenCL

Na figura 08 temos uma função chamada `compute_julia_pixel` que é definida como um kernel usando a palavra-chave `kernel`. Esse kernel é executado em paralelo por várias threads na GPU.

A função `compute_julia_pixel` realiza o processamento paralelo dos pixels de uma imagem, calculando os novos valores RGB com base nas coordenadas e em uma função auxiliar `compute_julia_pixel`. Os resultados são armazenados no array de saída `output_pixels`.

```

1  __kernel void compute_julia_pixel(__global int* input_pixels, __global uchar* output_pixels, const int largura, const int altura) {
2      int x = get_global_id(0);
3      int y = get_global_id(1);
4
5      // Check coordinates
6      if ((x < 0) || (x >= largura) || (y < 0) || (y >= altura)) {
7          return;
8      }
9
10     // Compute the index of the pixel in the input array
11     int index = y * largura + x;
12
13     // Extract the RGB values from the input pixel
14     int r = (input_pixels[index] & 0xFF0000) >> 16;
15     int g = (input_pixels[index] & 0x00FF00) >> 8;
16     int b = (input_pixels[index] & 0x0000FF);
17
18     // Compute the new RGB values for the output pixel using the compute_julia_pixel function
19     float tint_bias = 0.5;
20     unsigned char rgb[3];
21     int result = compute_julia_pixel(x, y, largura, altura, tint_bias, rgb);
22
23     // Write the new RGB values to the output pixel
24     output_pixels[index * 3] = rgb[0];
25     output_pixels[index * 3 + 1] = rgb[1];
26     output_pixels[index * 3 + 2] = rgb[2];
27 }
28

```

Figura 09 - Trecho do código onde é chamado o código OpenCL

### 3.3 Dificuldades

No desenvolvimento desse código foi passado por diversas dificuldades

- Configurar os argumentos para o kernel
- Construção do arquivo .cl
- Configuração do openCL
- Alocação de memória
- Cor do fractal

### 3.4 Resultado



Figura 07 - Fractal do código OpenCL

#### 4. Comparação

A tabela 01 apresenta uma comparação entre diferentes hosts e códigos que calculam o tempo de execução diretamente dentro do código. E na tabela 02 apresenta uma comparação entre diferentes hosts e códigos que o tempo é calculado via time do linux.

Tabela 01 - Comparação entre Host e códigos com tempo calculado dentro do código

| Host                          | Tempo de execução - OMP (segundos) | Tempo de execução - CUDA (segundos) | Tempo de execução - OpenCL (segundos) |
|-------------------------------|------------------------------------|-------------------------------------|---------------------------------------|
| cm1 (Intel i7-8700)           | 0,47                               | -                                   | -                                     |
| pos1(Intel i7-9700)           | 0,45                               | -                                   | -                                     |
| gpu1(Ryzen 7 2700 + RTX 3060) | 0,61                               | 0,0482                              | 0,000004                              |
| gpu2(Ryzen 7 2700 + GTX 1650) | 0,61                               | 0,1169                              | 0,000004                              |



Tabela 02 - Comparação entre Host e códigos com time do linux

| Host                          | Tempo de execução - OMP (segundos)    | Tempo de execução - CUDA (segundos)   | Tempo de execução - OpenCL (segundos) |
|-------------------------------|---------------------------------------|---------------------------------------|---------------------------------------|
| cm1 (Intel i7-8700)           | Real 0,815<br>User 1,918<br>Sys 0,07  | -                                     | -                                     |
| pos1(Intel i7-9700)           | Real 0,889<br>User 1,904<br>Sys 0,163 | -                                     | -                                     |
| gpu1(Ryzen 7 2700 + RTX 3060) | Real 1,002<br>User 2,620<br>Sys 0,092 | Real 0,365<br>User 0,057<br>Sys 0,192 | Real 0,862<br>User 0,625<br>Sys 0,127 |
| gpu2(Ryzen 7 2700 + GTX 1650) | Real 0,948<br>User 2,466<br>Sys 0,120 | Real 0,401<br>User 0,131<br>Sys 0,154 | Real 0,892<br>User 0,623<br>Sys 0,157 |

observação:

- Tempo real: o tempo total decorrido desde o início até a conclusão do comando, incluindo tempo gasto em tarefas como leitura e gravação em disco.
- Tempo do usuário (user): o tempo gasto pelo processador executando o código do comando.
- Tempo do sistema (sys): o tempo gasto pelo processador em tarefas do kernel relacionadas ao comando, como alocação de memória.

## 5. Conclusão

Com base nos dados coletados, podemos observar que a solução mais rápida em termos de tempo de execução é a utilização do CUDA na GPU1 (Ryzen 7 2700 + RTX 3060), com um tempo de apenas 0,0482 segundos. Em comparação, o tempo de execução utilizando OMP no cm1 (Intel i7-8700) foi de 0,47 segundos, sendo quase dez vezes maior que o tempo da solução mais rápida.

Além disso, podemos verificar que a eficiência das GPUs varia de acordo com a tecnologia utilizada. Enquanto a GPU1 (Ryzen 7 2700 + RTX 3060) apresentou um desempenho superior utilizando CUDA, a GPU2 (Ryzen 7 2700 + GTX 1650) apresentou um desempenho melhor utilizando OpenCL. Entretanto, mesmo utilizando a tecnologia mais eficiente para cada GPU, a

GPU1 ainda se mostrou mais rápida na execução da aplicação.

Em relação aos tempos real, de usuário e de sistema, podemos observar que a utilização das GPUs apresentou tempos mais longos tanto para o usuário quanto para o sistema, indicando que as GPUs demandam mais recursos de processamento nesses aspectos. No entanto, ainda assim, a utilização das GPUs se mostrou mais rápida na execução da aplicação em comparação aos hosts cm1 e pos1.

Em resumo, a utilização do CUDA na GPU1 (Ryzen 7 2700 + RTX 3060) se mostrou a solução mais eficiente para a execução da aplicação em questão, apresentando uma redução significativa no tempo de execução em comparação aos hosts cm1 e pos1. Porém é importante ressaltar que o tempo de execução não é a única métrica para determinar a eficiência de uma solução. Outros fatores, como consumo de energia, capacidade de processamento paralelo e requisitos específicos da aplicação, também devem ser considerados ao avaliar a eficiência de uma determinada solução.