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 opm 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.

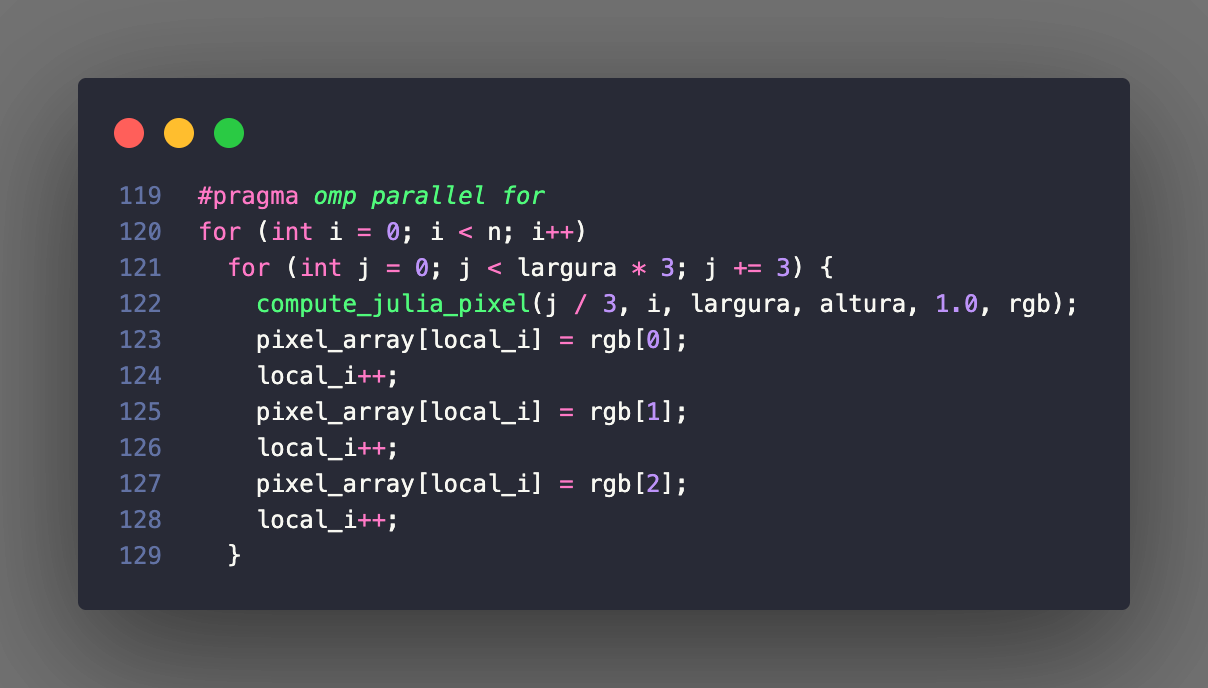
****

Figura 01 - Trecho do código OpenMP

**1.1 Dificuldades**

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

**1.2 Resultado**

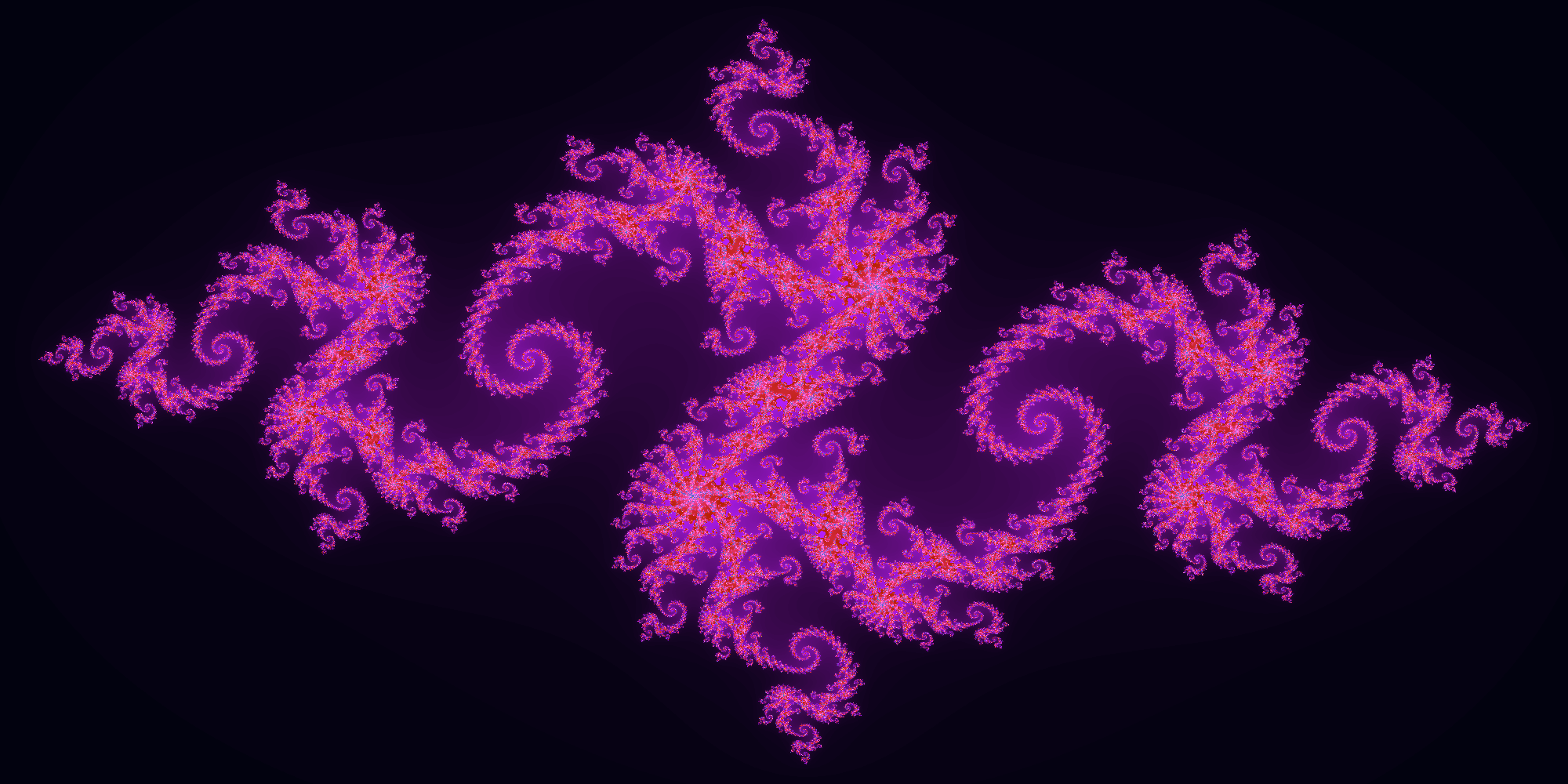


Figura 02 - Fractal do código OpenMP

1. **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.



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.



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.

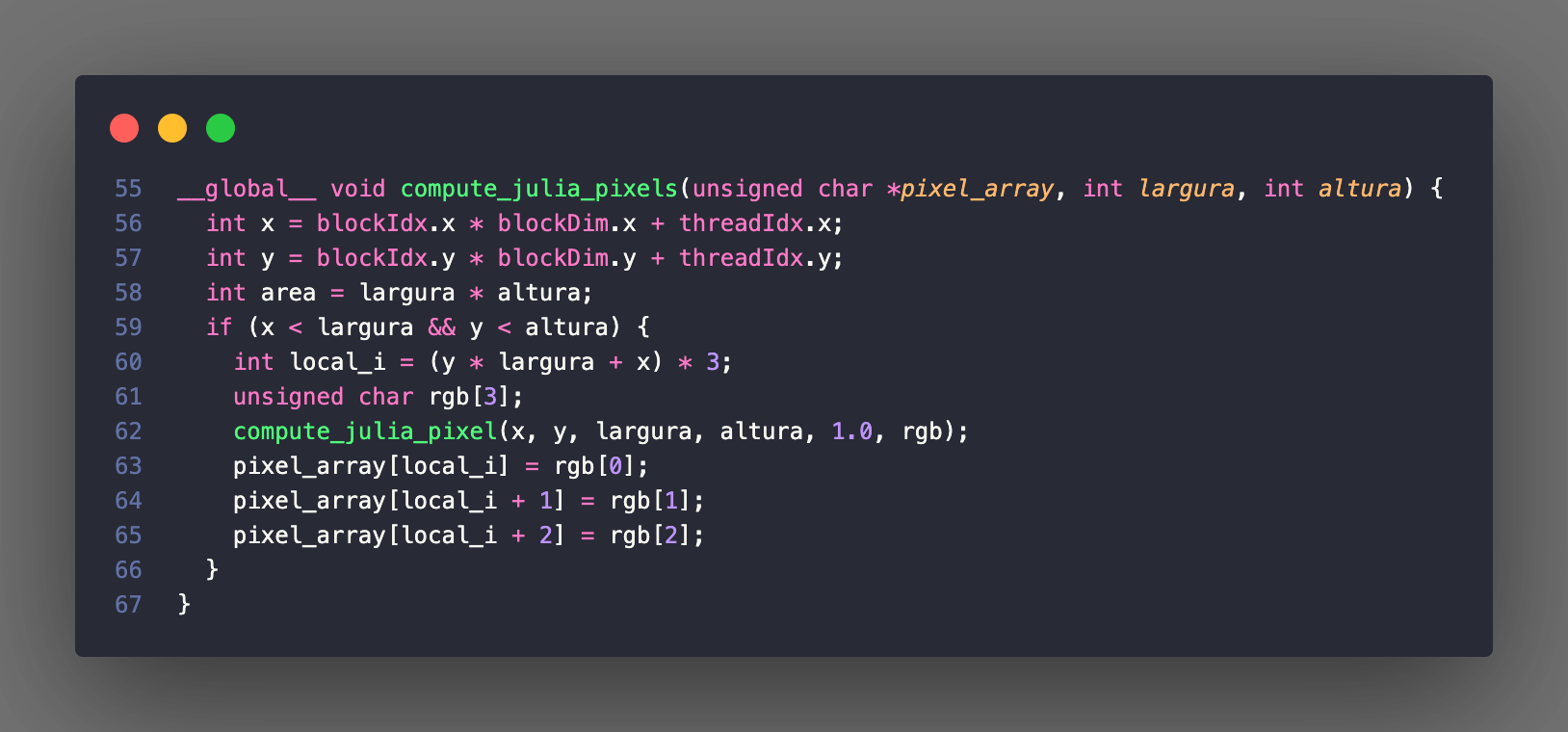


Figura 05 - Trecho do código que é paralelizado.

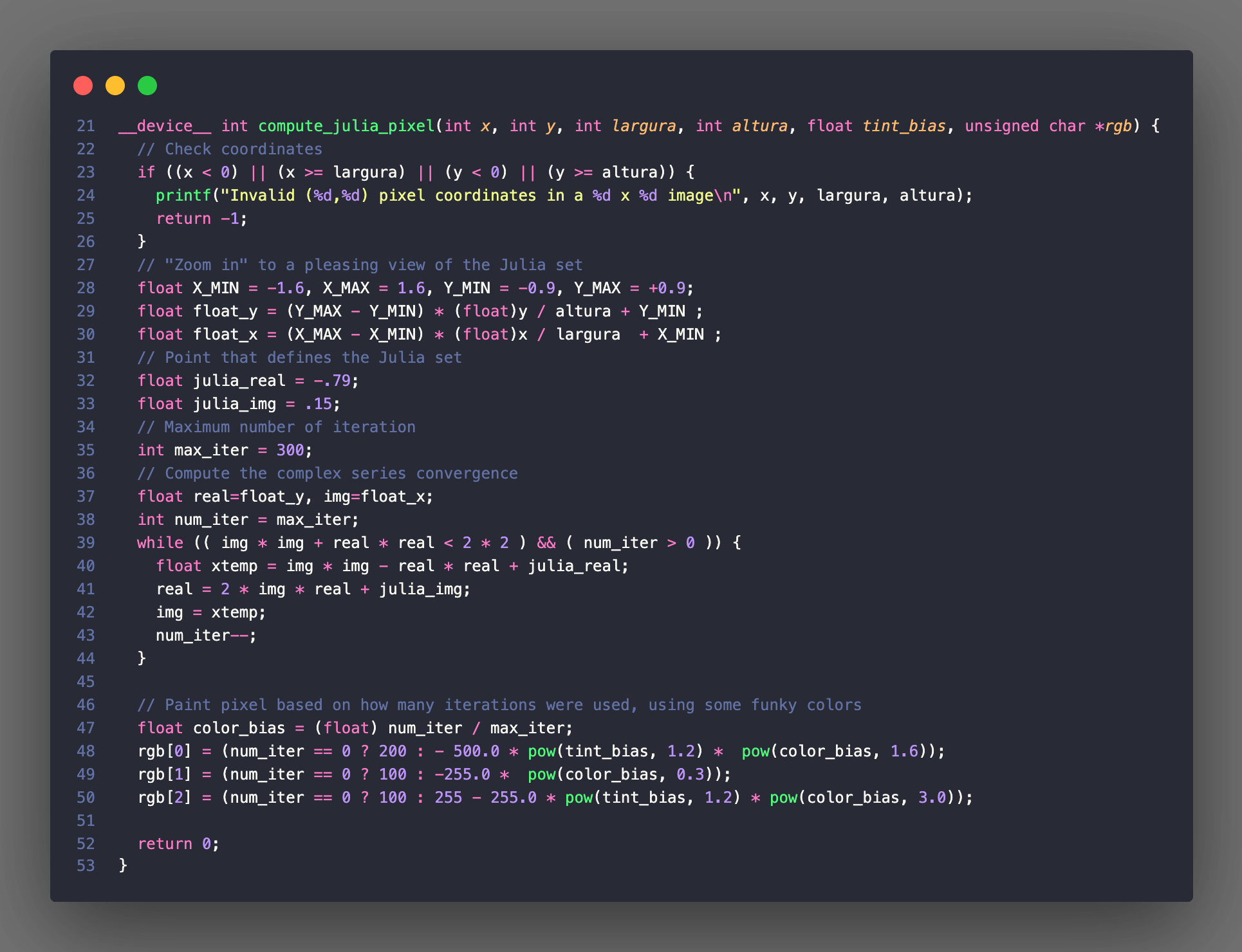


Figura 06 - Trecho do código CUDA com a função compute\_julia\_pixel

**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
* Inclusão do tempo de execução

**2.3 Resultados**

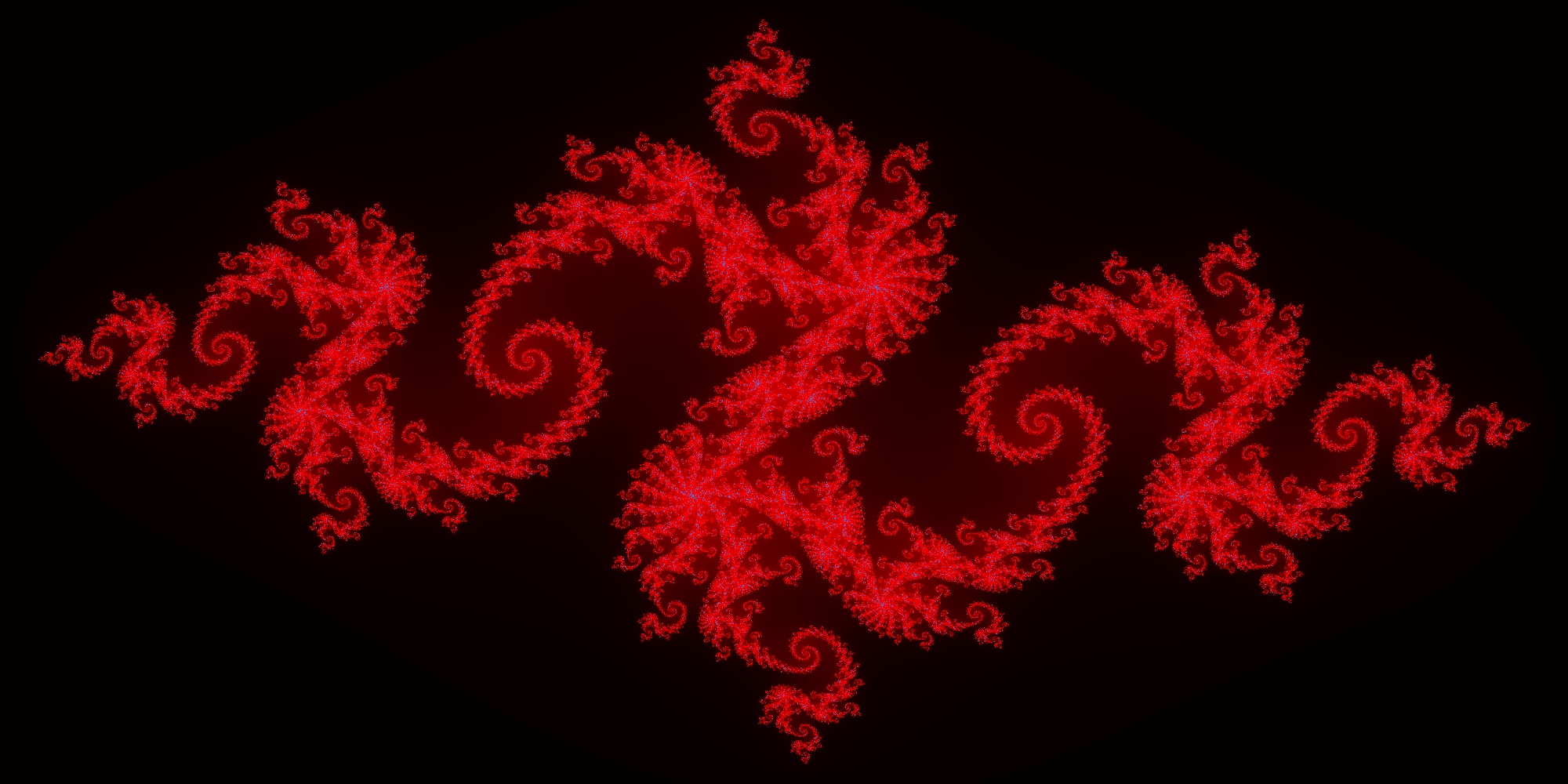


Figura 07 - Fractal do código CUDA

1. **OpenCL**

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

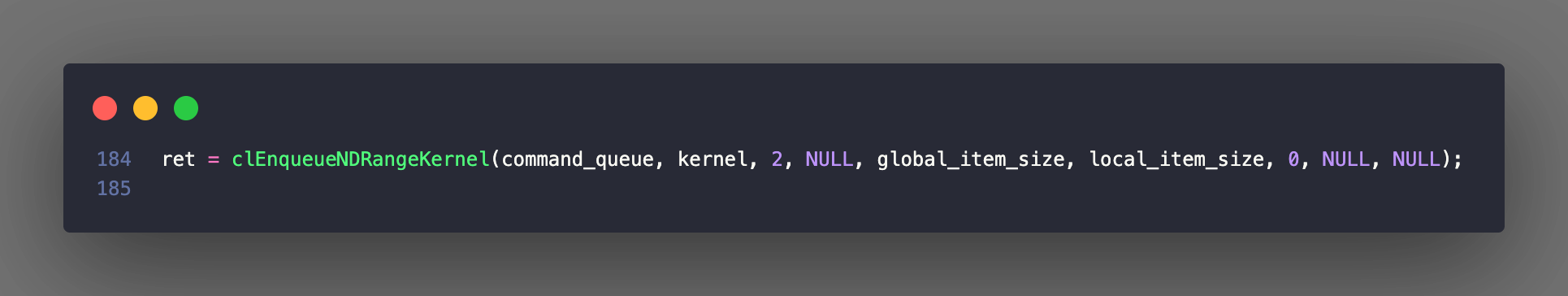


Figura 08 - Trecho do código onde é chamado o código OpenCL

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.



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 kernel
* Alocação de memória
* Inclusão do tempo de execução

**3.4 Resultado**

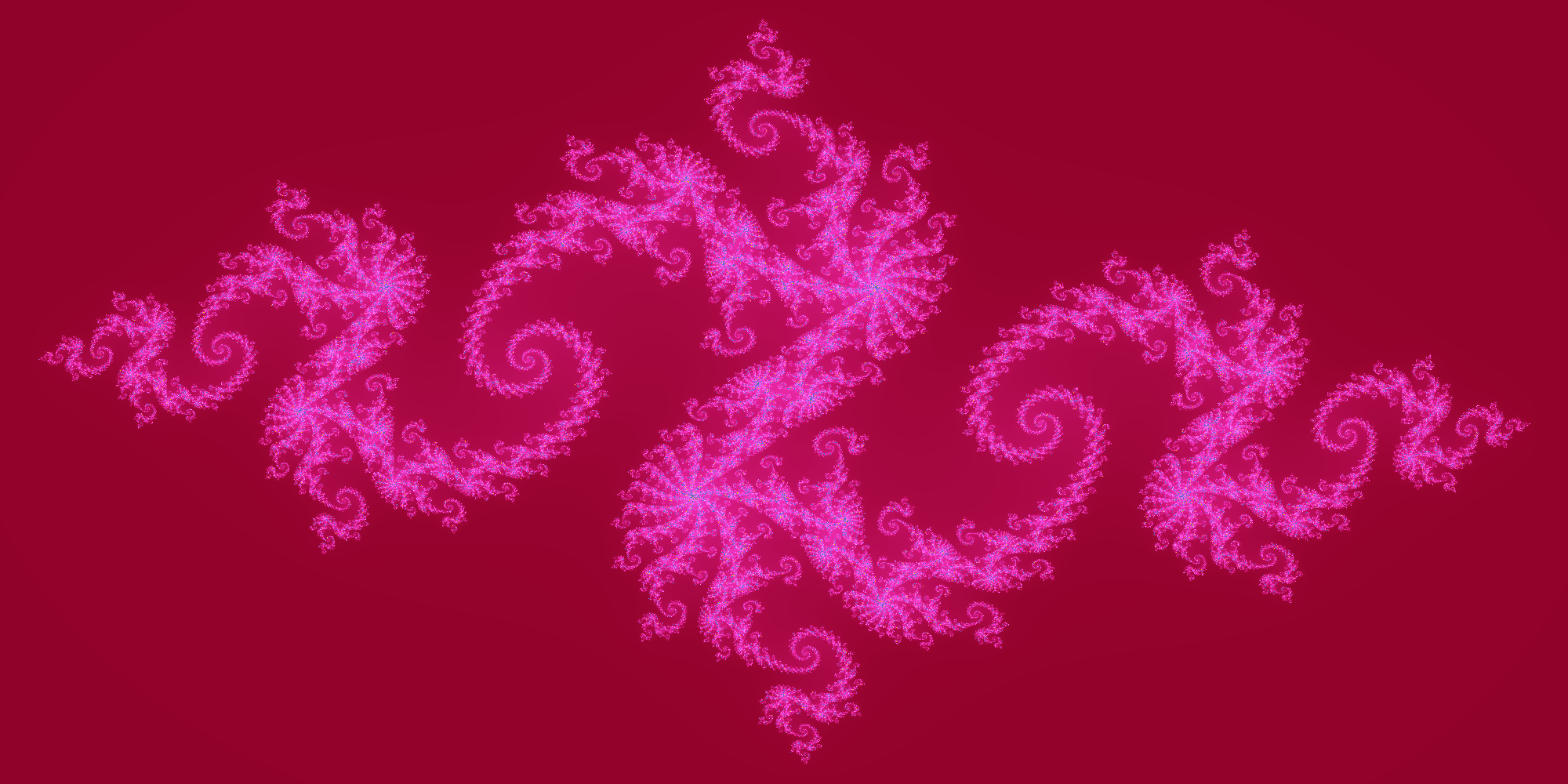


Figura 07 - Fractal do código OpenCL

**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  User 1,918  Sys 0,07 | - | - |
| pos1(Intel i7-9700) | Real 0,889  User 1,904  Sys 0,163 | - | - |
| gpu1(Ryzen 7 2700 + RTX 3060) | Real 1,002  User 2,620  Sys 0,092 | Real 0,365  User 0,057  Sys 0,192 | Real 0,862  User 0,625  Sys 0,127 |
| gpu2(Ryzen 7 2700 + GTX 1650) | Real 0,948  User 2,466  Sys 0,120 | Real 0,401  User 0,131  Sys 0,154 | Real 0,892  User 0,623  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.