UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

BRUNO DE AVILA

PARALELIZAÇÃO DE APLICAÇÕES DE CÁLCULO DE FRACTAIS PARA GPU UTILIZANDO NUMBA

CAXIAS DO SUL

2022

BRUNO DE AVILA

PARALELIZAÇÃO DE APLICAÇÕES DE CÁLCULO DE FRACTAIS PARA GPU UTILIZANDO NUMBA

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. André Luis Martinotto

CAXIAS DO SUL

2022

BRUNO DE AVILA

PARALELIZAÇÃO DE APLICAÇÕES DE CÁLCULO DE FRACTAIS PARA GPU UTILIZANDO NUMBA

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado(a) em 01/12/2022

BANCA EXAMINADORA

Prof. Dr. André Luis Martinotto
Universidade de Caxias do Sul - UCS

Prof. Dr. Cláudio Antônio Perottoni
Universidade de Caxias do Sul - UCS

Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

RESUMO

O interesse em fractais reside na grande quantidade de situações práticas em que esses objetos abstratos são utilizados. Entre outras aplicações, pode-se citar a área de mineralogia, onde são usados para o cálculo da densidade de minerais, e na área da oceanografia, onde são utilizados para auxiliar na análise das extensões litorâneas. Além disso, observa-se um crescente interesse pela arte fractal, principalmente, nas áreas de computação gráfica, animações e outros tipos de mídia. Os fractais comumente são gerados utilizando funções matemáticas, cujos resultados são transformados em imagens. No entanto, para a geração de fractais cada vez mais complexos e detalhados, torna-se necessário um alto poder computacional. Desta forma, neste trabalho foi realizado um estudo sobre a paralelização de cálculos de fractais utilizando unidades de processamento gráfico. Mais especificamente foram paralelizados três fractais que são gerados a partir de relações de recorrência, os fractais de Julia, de Mandelbrot 2D e 3D; e um fractal aleatório, o fractal da árvore browniana. As implementações foram desenvolvidas utilizando a linguagem de programação Python e foram paralelizadas para GPU utilizando a biblioteca Numba. Para avaliação, foi realizada uma comparação entre os tempos de execução das implementações, em que os resultados que apresentaram o menor tempo de execução foram os executados na GPU chegando a serem 858 vezes mais rápidos que os executados na CPU. A implementação feita em Python apresentou um tempo de execução muito maior quando comparada aos outros métodos implementados.

Palavras-chave: Fractal. Paralelização. Unidades de Processamento Gráfico.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

GPU	*Graphics Processing Unit*
CPU	*Central Processing Unit*
SISD	*Single Instruction Single Data*
SIMD	*Single Instruction Multiple Data*
MISD	*Multiple Instruction Single Data*
MIMD	*Multiple Instruction Multiple Data*
CES	*Consumer Electronics Show*
CEO	*Chief Executive Officer*
ULA	Unidade Lógica e Aritmética
SM	*Streaming Multiprocessors*
DRAM	*Dynamic Random Access Memory*
NVCC	NVIDIA C *Compiler*
PTX	*Parallel Thread eXecution*
MPI	*Message Passing Interface*
JIT	*Just in Time*
RAM	*Random Access Memory*
SSD	*Solid State Drive*

SUMÁRIO

1 INTRODUÇÃO

A primeira utilização do termo fractal é datada de 1975, quando o matemático Benoit Mandelbrot utilizou a palavra para descrever uma geometria que ele havia desenvolvido para representar formas da natureza (ASSIS *et al.*, 2008). Por definição, um fractal é um objeto geométrico que pode ser dividido em partes, onde cada uma delas é semelhante ao objeto original. Geralmente, um fractal é gerado através de um padrão repetido, ou seja, através de um processo iterativo (PEITGEN; SAUPE, 2011).

Os fractais matemáticos apresentam características que são similares às formas encontradas na natureza, tais como bacias hidrográficas, correntes sanguíneas, espécies de flores, etc. Desta forma, o estudo da geometria fractal é considerado uma área de grande interesse, uma vez que transcende o campo da matemática, auxiliando na compreensão da geometria de formas e de fenômenos da natureza (FERNANDES, 2010).

O processo computacional para a geração de um fractal apresenta um elevado custo computacional, visto que é um processo iterativo e que envolve diversos fatores, como o número de iterações e a dimensão do conjunto (ASSIS *et al.*, 2008). Assim, torna-se atrativa a utilização de técnicas que possibilitem acelerar o processo de geração de um fractal. Entre essas técnicas, uma que tem recebido destaque é a programação paralela (BAKER, 2011). A programação paralela consiste em dividir um problema em tarefas menores que podem ser executadas simultaneamente (LORIN, 1990).

Dentre as arquiteturas paralelas disponíveis, há uma que está se sobressaindo nos últimos anos: as unidades de processamento gráfico (GPU, do inglês *Graphics Processing Unit*). O uso das GPUs vem crescendo rapidamente devido, principalmente, ao elevado poder de processamento que as GPUs apresentam quando comparadas à unidade central de processamento (CPU, do inglês *Central Processing Unit*). De fato, quando se compara o número de unidades de processamento, observa-se que uma GPU apresenta um número muito maior de unidades de processamento que uma CPU, o que possibilita a execução simultânea de um número maior de instruções (SETH, 2021).

Uma das linguagens de programação que mais cresce em quantidade de usuários e utilização é a linguagem Python (JOURY, 2020). Esse aumento é decorrente devido, principalmente, à facilidade de utilização e ao grande número de pacotes disponíveis para as mais variadas aplicações (THE..., 2022). No que se refere à geração de fractais, destacam-se o módulo Pillow, que é utilizado para a manipulação de imagens (PILLOW, 2022); e a biblioteca NumPy, que é uma biblioteca para a computação científica (NUMPY, 2022). Além disso, para a utilização de GPUs encontram-se disponíveis as bibliotecas CUDA Python e Numba, que possibilitam a utilização do *framework* CUDA na linguagem de programação Python (LEVINAS, 2021).

Dentro desse contexto, neste Trabalho de Conclusão de Curso foi realizado um estudo sobre a utilização da linguagem de programação Python para a geração de fractais 2D e 3D. As implementações desenvolvidas neste trabalho foram paralelizadas de forma a serem executadas em uma arquitetura de GPU. Para a paralelização destas implementações foi utilizada a biblioteca Numba.

1.1 OBJETIVOS

O principal objetivo deste trabalho consistiu no desenvolvimento de aplicações paralelas em GPU para a geração de fractais 2D e 3D. Para o desenvolvimento, foram utilizadas a linguagem de programação Python e a biblioteca Numba. Para que o objetivo principal fosse atingido, os seguintes objetivos específicos foram realizados:

- Definição dos fractais 2D e 3D a serem implementados;
- Desenvolvimento das implementações sequenciais dos fractais;
- Paralelização, em GPU, das implementações desenvolvidas;
- Comparação dos tempos de execução entre as implementações sequenciais e paralelizadas para GPU.

1.2 ESTRUTURA DO TRABALHO

O trabalho apresentado está organizado da seguinte maneira:

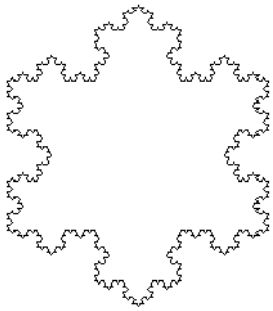
- Neste Capítulo foi realizada uma breve introdução do trabalho, apresentando sua motivação e objetivos;
- No Capítulo 2 são apresentados os principais conceitos relacionados aos fractais e suas aplicações;
- No Capítulo 3 é realizada uma breve conceituação de arquiteturas paralelas. Uma ênfase maior é dada para a arquitetura de GPUs da empresa NVIDIA, uma vez que essa foi a arquitetura utilizada para o desenvolvimento deste trabalho;
- No Capítulo 4 são definidos os fractais que foram paralelizados neste trabalho. Além disso, são apresentadas as implementações sequenciais dos mesmos;
- No Capítulo 5 é apresentada a biblioteca Numba e as implementações paralelas dos fractais escolhidos;
- No Capítulo 6 são apresentados os testes e os resultados obtidos;

- Por fim, no Capítulo 7 são apresentadas as considerações finais do trabalho e as sugestões de trabalhos futuros.

2 FRACTAIS

O matemático francês Benoit Mandelbrot foi o primeiro a utilizar um computador para a geração de fractais. Desde então, os estudos avançaram significativamente, sendo criado até um ramo da matemática conhecido como geometria fractal. Os fractais são compostos por objetos que possuem partes semelhantes, onde os padrões de uma forma se repetem, mas em uma escala menor. Como exemplo, pode-se citar o fractal do floco de neve, também conhecido como curva de Koch, que é apresentado na Figura 1.

Figura 1 – Curva de Koch ou fractal do Floco de Neve



Fonte: Filho (2002).

A geometria fractal é um campo que se apresenta como uma alternativa a problemas em que a geometria euclidiana não pode ser utilizada para representar uma forma ou objeto. De fato, a geometria euclidiana não se mostra adequada para representar inúmeras formas existentes na natureza, uma vez que ela é baseada somente na utilização de círculos, triângulos, esferas, icosaedros, retângulos, etc (ASSIS *et al.*, 2008). Por exemplo, um modelo euclidiano representado por círculo, quando ampliado se torna cada vez mais parecido com uma linha reta. Um fractal, em contrapartida, quando ampliado fica cada vez mais detalhado.

Os fractais são utilizados em diversas áreas do conhecimento. Por exemplo, na área da saúde são utilizados em estudos relacionados ao desenvolvimento de tumores cancerígenos (KREUTZ; KEPLER; STEIN, 2019); na área de mineralogia, para calcular a densidade de minerais (ZHERU; HUAHAI; CHENG, 2001); na biologia, por sua vez, para a análise da rugosidade de corais e fungos (BRADBURY; REICHELT; GREEN, 1984); e na indústria, para a detecção automática de defeitos em produtos têxteis (PROENCA; CONCI; SEGENREICH, 1999). Uma outra área em que se pode notar a aparição dos conceitos de fractais é a da Teoria do Caos[1], em que as suas equações são utilizadas para descrever fenômenos, que ainda que pareçam aleatórios, obedecem a certas regras (FEY; ROSA, 2012).

[1] Pequenas mudanças no início de um evento podem trazer consequências desconhecidas no futuro

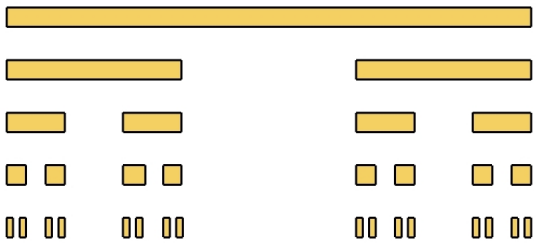
2.1 PROPRIEDADES DOS FRACTAIS

Os fractais apresentam propriedades que os distinguem de outras formas, que são: a auto-similaridade, a complexidade infinita e a dimensionalidade. A auto-similaridade se refere à presença de uma repetição parcial de uma figura em uma escala diferente da original. A complexidade infinita implica que um fractal é um objeto recursivo, ou seja, dentro dele você encontrará outras instâncias que são idênticas ou similares a ele. No que se refere à dimensionalidade, um fractal se difere dos modelos euclidianos pois em um modelo euclidiano a dimensão é um número inteiro, enquanto em um fractal a dimensão pode ser um valor fracionário.

2.1.1 Auto-similaridade

Uma característica que distingue os fractais de outras formas geométricas é a sua auto-similaridade. Os fractais mais conhecidos, como o floco de neve de Koch (Figura 1), apresentam essa característica de forma clara, uma vez que todos os flocos são idênticos, porém em escalas diferentes. Outro fractal que apresenta essa característica é o fractal gerado pelo conjunto de Cantor. Como pode ser observado na Figura 2, todas as linhas que formam o fractal são iguais, porém com comprimentos diferentes.

Figura 2 – Conjunto de Cantor

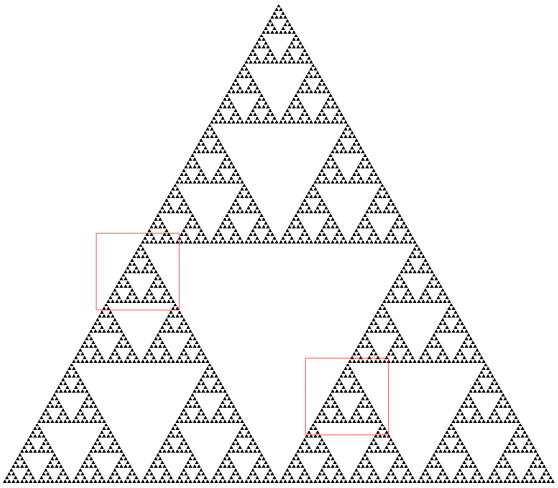


Fonte: Sedrez (2009).

Nem todos os fractais apresentam o mesmo padrão de auto-similaridade. Dessa forma, de acordo com essa característica, os fractais podem ser divididos em:

- Auto-similaridade exata: onde cada parte do fractal é idêntica, independentemente da escala. Como exemplo, pode-se citar os fractais que são gerados por regras de substituição. Um exemplo de um fractal que apresenta uma auto-similaridade exata é o fractal Triângulo de Sierpinski, que pode ser observado na Figura 3.

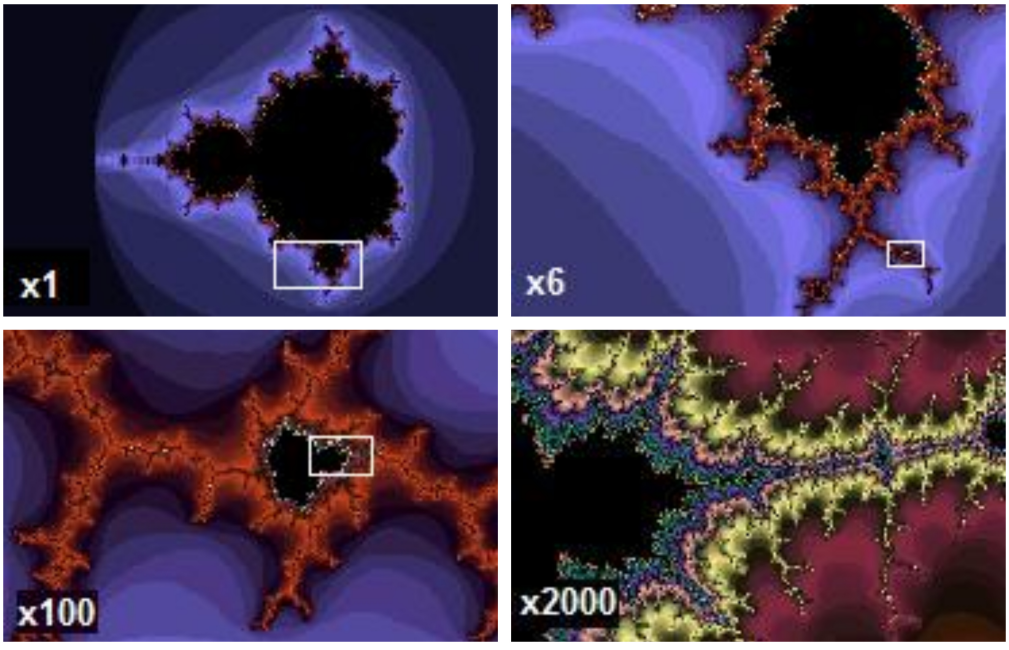
Figura 3 – Triângulo de Sierpinski destacado em partes idênticas



Fonte: O autor (2022).

- Quase-auto-similaridade: o fractal reproduz o mesmo padrão em escalas diferentes, porém com algumas distorções. Como exemplo, pode-se citar os fractais gerados a partir de relações de recorrência, tal qual o conjunto de Mandelbrot. Quando esse fractal é ampliado, ele mantém o mesmo padrão do original, como pode ser observado na Figura 4. No entanto, a partir de uma análise mais cuidadosa pode-se observar que esse apresenta pequenas variações em determinados pontos.

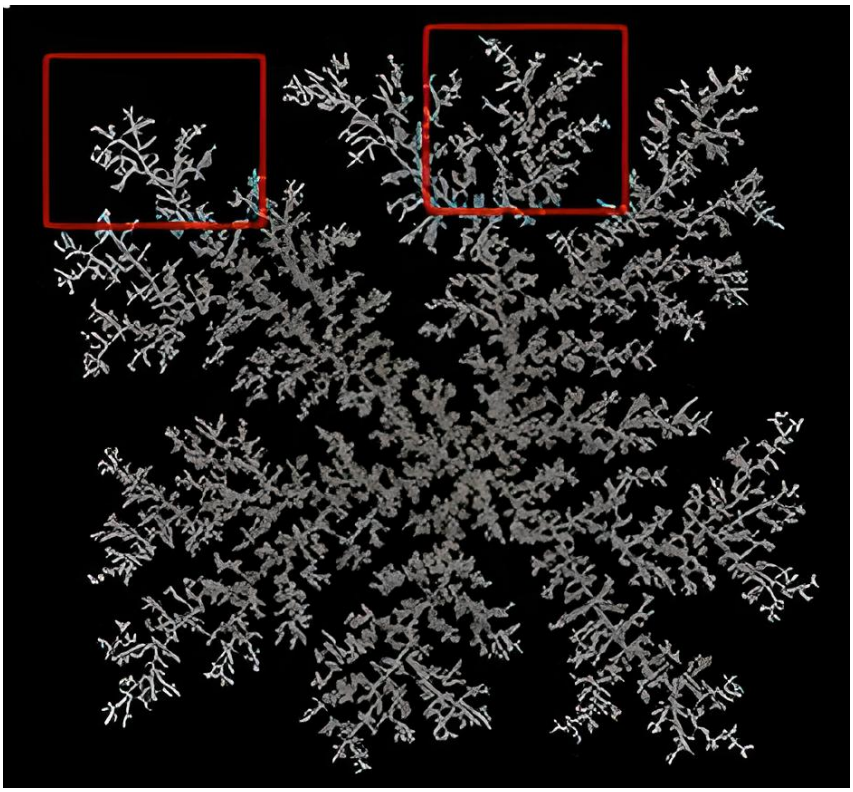
Figura 4 – Conjunto de Mandelbrot ampliado



Fonte: Ferguson (2012).

- Auto-similaridade estatística: é a forma menos evidente de auto-similaridade. Nesse caso o fractal apresenta apenas semelhanças quando ampliado. Como exemplo, pode-se citar os fractais aleatórios, como o da árvore browniana, que é apresentado na Figura 5. Como pode ser observado, nesse caso existe apenas uma semelhança entre os ramos da árvore.

Figura 5 – Árvore browniana destacada pelas semelhanças



Fonte: Adaptado de Alfio (2003).

2.1.2 Complexidade infinita

Os fractais são gerados a partir de processos iterativos ou recursivos, que são infinitos. Desta forma, nunca será possível representar completamente um fractal, uma vez que esse possui uma quantidade de detalhes infinita, ou seja, sempre haverá reentrâncias e saliências cada vez menores à medida que o fractal for ampliado. Essa propriedade é uma característica existente somente em fractais ideais, uma vez que os fractais reais possuem limitações de escala (FUZZO, 2009).

2.1.3 Dimensionalidade

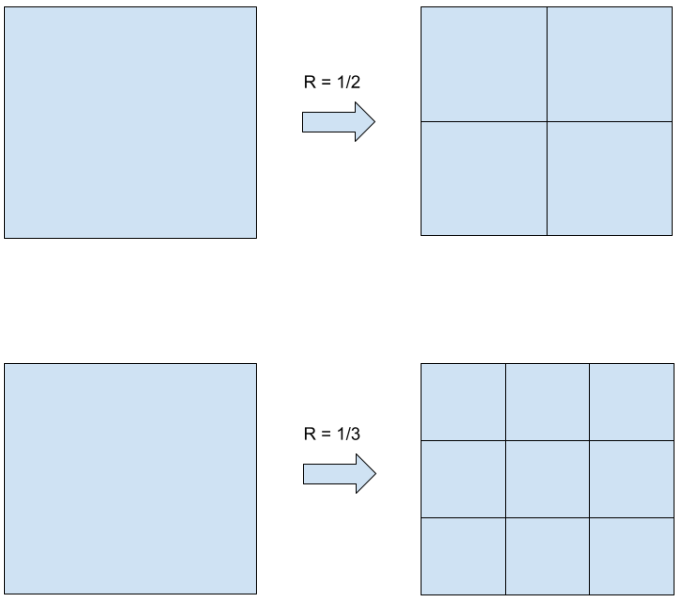
Na geometria euclidiana, o conceito de dimensão consiste em um número inteiro que é utilizado para caracterizar um objeto euclidiano. Por exemplo, um ponto apresenta uma dimensão igual a 0; uma reta possui uma dimensão igual a 1; um plano possui uma dimensão 2; e um volume possui uma dimensão 3. No entanto, a geometria euclidiana não é suficiente para representar um objeto fractal, já que esses possuem uma estrutura muito fina, com escalas arbitrariamente pequenas e irregulares. Dessa forma, para a representação de fractais é utilizado um conceito chamado de dimensão espacial ou dimensão fractal. A dimensão espacial refere-se ao espaço que um objeto ocupa, podendo ser representada por números inteiros ou fracionários (ARSIE, 2007).

A dimensão espacial pode ser calculada de diversas maneiras, sendo que entre as formas mais utilizadas estão a dimensão por auto-similaridade e a dimensão por contagem de caixas (ARSIE, 2007). O processo de cálculo através da auto-similaridade é simples, sendo que ele é aplicado em fractais onde a auto-similaridade é exata. Considerando-se uma reta, que possui dimensão $D = 1$, quando ela tem sua extensão dobrada obtém-se 2 cópias idênticas à original. No caso de um quadrado de dimensão $D = 2$, quando a sua extensão é dobrada obtém-se 4 cópias. Por fim, dobrando a extensão de um cubo de dimensão $D = 3$, obtém-se 8 cópias da figura original. Da mesma forma, a partir da divisão de uma figura, através do fator de redução R obtém-se N cópias por meio da relação

$$N = \frac{1}{R^D} \tag{2.1}$$

onde D é a dimensão espacial, R é fator de redução e N é o número de cópias da figura. Por exemplo, como pode ser observado na Figura 6, considerando-se um quadrado ($D = 2$), se a extensão deste for dividida em 2 partes, tem-se um fator $R = \frac{1}{2}$. Desta forma, tem-se que o número de cópias N é igual $N = 4$. Já através da divisão da extensão deste por 3 tem-se um fator $R = \frac{1}{3}$ e, consequentemente, um número de cópias $N = 9$.

Figura 6 – Razão do quadrado



Fonte: O Autor (2022).

Isolando-se a variável D, tem-se que a dimensionalidade espacial pode ser calculada através de

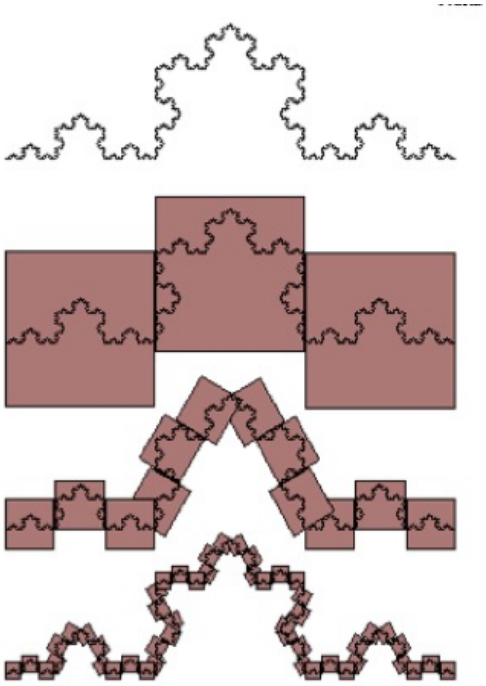
$$D = \frac{logN}{log\frac{1}{R}}. \tag{2.2}$$

Por exemplo, considerando-se o conjunto de Cantor (Figura 2), onde cada segmento é dividido em duas partes ($N = 2$) e que cada segmento é obtido utilizando um fator de redução $R = 1/3$, tem-se que a dimensão espacial D é

$$D = \frac{log\ \ 2}{log\ \ 1/(1/3)} = 0,63 \tag{2.3}$$

Para o cálculo da dimensão em fractais que não possuem auto-similaridade exata, pode ser utilizado o método da dimensão por contagem de caixas, que consiste em calcular a dimensão espacial do fractal cobrindo a figura com uma malha de quadrados com lados de dimensão r. Após, é realizada uma contagem do número N de quadrados que possuem em seu interior ao menos um ponto do fractal, ou seja, conta-se o número $N(r)$ de caixas necessárias para que se cubra todo o objeto. Esse processo é realizado iterativamente reduzindo o lado r do quadrado.

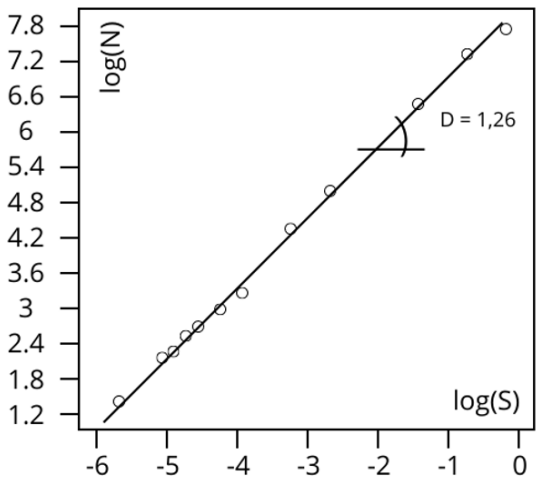
Figura 7 – Redução do lado do quadrado



Fonte: Adaptado de Pilgrim e Taylor (2018).

Após, um gráfico $log(N(r))$ em função de $log(1/r)$ é gerado, sendo que a inclinação da reta que se ajusta aos pontos corresponde a dimensão fractal. Na Figura 8 tem-se o gráfico correspondente ao exemplo apresentado na Figura 7. Nesse caso, a dimensão fractal por contagem de caixas corresponde a $1,26$.

Figura 8 – Cálculo da dimensão pelo método de contagem



Fonte: Adaptado de Sangiorgi, Collop e Thom (2003).

2.2 CATEGORIAS DE FRACTAIS

Os fractais podem ser classificados em três categorias, as quais dependem do modo como o fractal é formado ou gerado. De acordo com FUZZO (2009), os fractais podem ser divididos em: sistemas de funções iteradas, também chamados de fractais geométricos; fractais definidos por uma relação de recorrência, também chamados de fractais de fuga do tempo; e fractais aleatórios.

2.2.1 Sistema de funções iteradas

Os fractais de funções iteradas ou geométricos são gerados por transformações geométricas simples do próprio objeto. Nessa categoria encontram-se os fractais que possuem autosimilaridade exata, como o Conjunto de Cantor, a Curva de Koch e o Triângulo de Sierpinski (MASSAGO, 2010).

2.2.1.1 Conjunto de Cantor

O conjunto de Cantor, também conhecido como Poeira de Cantor, tem esse nome devido ao matemático russo George F. L. Philipp Cantor (1845 – 1918), que é conhecido por ter elaborado a teoria dos conjuntos moderna (FUZZO, 2009). O conjunto de Cantor é criado a partir de um conjunto de passos bem definidos. Primeiramente, é gerada uma reta de comprimento unitário. No segundo passo, essa reta é dividida em três partes iguais, sendo retirada a parte central. Em seguida, o segundo passo é realizado novamente e de forma sucessiva. Na Figura 9 é apresentado o processo de geração do Conjunto de Cantor.

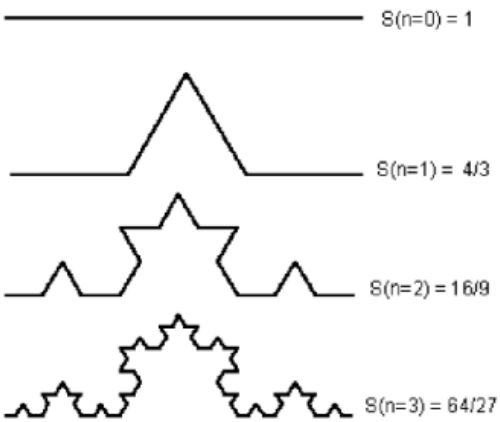
Figura 9 – Conjunto de Cantor

Fonte: Paula (2017).

2.2.1.2 Curva de Koch

A curva de Koch é um fractal que se assemelha a um floco de neve e foi criado pelo matemático Niels Fabian Helge von Koch, em 1904. Esse fractal tem início com um segmento de reta que é dividido em três partes, sendo retirada a parte central. Entretanto, neste caso, no lugar da parte retirada é colocado um triângulo equilátero sem a base, resultando em quatro novos segmentos de reta. Esse processo é repetido indefinidamente para todos os novos segmentos de reta que foram gerados, como pode ser observado na Figura 10.

Figura 10 – Curva de Koch

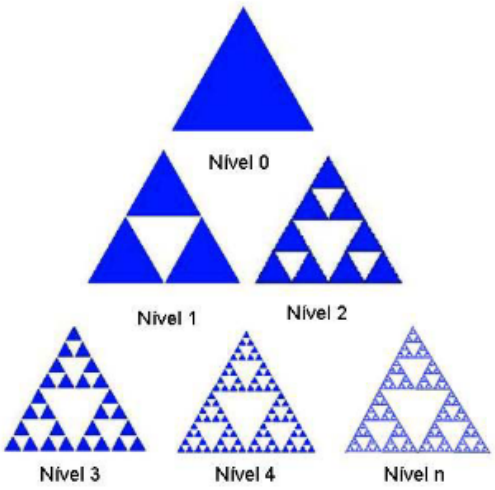


Fonte: Adaptado de Assis *et al.* (2008).

2.2.1.3 Triângulo de Sierpinski

O Triângulo de Sierpinski leva o nome do matemático polonês Waclaw Sierpinski, que foi o primeiro a descrever esse fractal em 1915. A construção do triângulo de Sierpinski é realizada a partir de um triângulo totalmente preenchido. Após, são definidos os pontos médios de cada um dos três segmentos que delimitam esse triângulo. A partir da ligação dos três pontos médios, obtém-se quatro triângulos congruentes, que possuem um lado de tamanho igual à metade do lado do triângulo original. Posteriormente, o triângulo central é removido, restando apenas três novos triângulos. Esse procedimento é repetido indefinidamente com os triângulos restantes, como pode ser observado na Figura 11.

Figura 11 – Triângulo de Sierpinski



Fonte: Adaptado de Pallesi (2007).

2.2.2 Fractais definidos por uma relação de recorrência

Esse tipo de fractal é definido por uma relação de recorrência em cada ponto do espaço (tal como um plano complexo). Os fractais mais conhecidos nessa categoria são o Conjunto de Mandelbrot e o Conjunto de Julia.

2.2.2.1 Conjunto de Mandelbrot

Os estudos sobre o conjunto de Mandelbrot foram iniciados pelo matemático francês Pierre Fatou em 1905, que estudou processos recursivos do tipo

$$\begin{aligned} z_0 &= 0 \\ z_{n+1} &= z_n^2 + c \end{aligned} \tag{2.4}$$

onde c é um conjunto de números complexos do tipo $c = x + yi$. Assim, para cada ponto (x, y) é obtido um ponto c no plano complexo, onde a sequência se expande, sendo que o valor da sequência em uma iteração é obtido a partir do valor obtido na iteração anterior. Por exemplo, considerando-se um ponto $(0, 1)$ onde o valor da constante $c = i$ tem-se

$$\begin{aligned} z_0 &= 0 \\ z_1 &= 0^2 + i = i \\ z_2 &= (i)^2 + i = -1 + i \\ z_3 &= (-1 + i)^2 + i = -i \\ z_4 &= (-i)^2 + i = -i + i \\ z_5 &= (-i + i)^2 + i = -i \end{aligned}$$

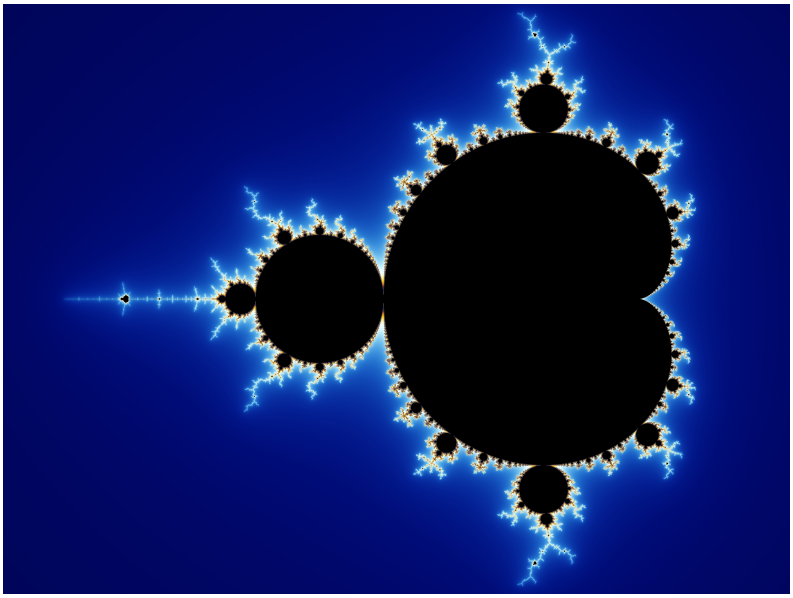
Já se for considerado um ponto $(0, 2)$ onde o valor de $c = 2i$ tem-se que

$$\begin{aligned} z_0 &= 0 \\ z_1 &= 0^2 + 2i = 2i \\ z_2 &= (2i)^2 + 2i = -4 + 2i \\ z_3 &= (-4 + 2i)^2 + 2i = 12 - 14i \\ z_4 &= (12 - 14i)^2 + 2i = -52 - 334i \end{aligned}$$

Um problema que Fatou enfrentou é que dependendo do valor de c, a sequência diverge para o infinito. Uma vez que não existiam computadores na época, Fatou tentou calcular essas sequências manualmente, porém não obteve sucesso, visto que muitos pontos tinham diversas iterações antes de tenderem ao infinito ou não.

Posteriormente, em 1979, Mandelbrot com auxílio de computadores, obteve os valores de c nos quais a série não tende ao infinito e esses são os valores que compõem o Conjunto de Mandelbrot (REIS, 2016). Assim, os pontos que vão formar o fractal de Mandelbrot são somente aqueles valores que a série não tende ao infinito. A Figura 12 apresenta o conjunto de Mandelbrot, onde os pontos pretos correspondem aos pontos do conjunto.

Figura 12 – Conjunto de Mandelbrot



Fonte: Beyer (2013).

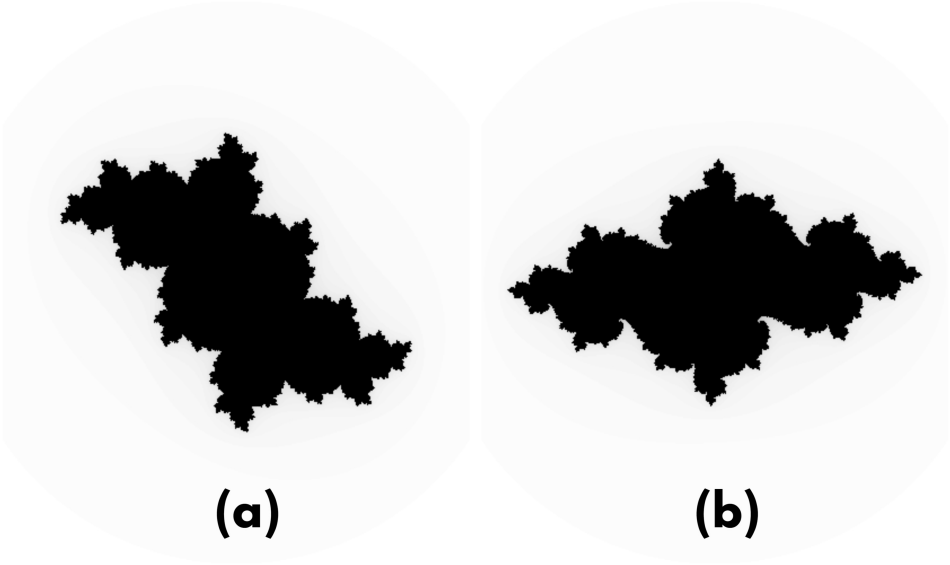
2.2.2.2 Conjunto de Julia

O conjunto de Julia é um fractal que recebeu esse nome em homenagem ao matemático francês Gaston Julia, que descobriu as propriedades básicas desse conjunto. O conjunto de Julia utiliza a mesma equação que o conjunto de Mandelbrot. No entanto, na construção do fractal, em vez do valor $z_0 = 0$, utiliza-se para z_0 o valor do ponto a ser iterado no espaço complexo e um número complexo para a constante c, por exemplo, $c = -0,12 + 0,6i$. A Equação 2.5 apresenta a equação correspondente a esse fractal

$$c = -0,12 + 0,6i$$
$$z_{n+1} = z_n^2 + c$$

Dessa forma, considerando-se diferentes valores para c, geram-se diferentes fractais, ou seja, uma nova imagem é formada. Por exemplo, a Figura 13 (a) apresenta o conjunto de Julia gerado a partir de $c = -0,12 + 0,6i$. Já a Figura 13 (b) apresenta o fractal resultante utilizando-se um valor de $c = -0,7 - 0,1i$.

Figura 13 – Conjunto de Julia



Fonte: O autor (2022).

2.2.2.3 Mandelbulb

O fractal conhecido por Mandelbulb é a representação do conjunto de Mandelbrot em um espaço de três dimensões. Ele foi descoberto por Daniel White e Paul Nylander e é gerado utilizando um sistema de coordenadas esféricas. O fractal de Mandelbulb é representado através da equação

$$v = v^n + c \tag{2.5}$$

onde v é um vetor em R^3. O valor de v^n pode ser calculado através de

$$v^n = r^n(seno(\theta * n) * cos(\phi * n), seno(\theta * n) * seno(\phi * n), cos(\theta * n)) \tag{2.6}$$

onde

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \theta &= arctan\frac{\sqrt{x^2 + y^2}}{z} \\ \phi &= arctan\frac{y}{x} \end{aligned} \tag{2.7}$$

O Algoritmo 1 apresenta o pseudocódigo que é utilizado para o cálculo do novo valor de v a cada iteração.

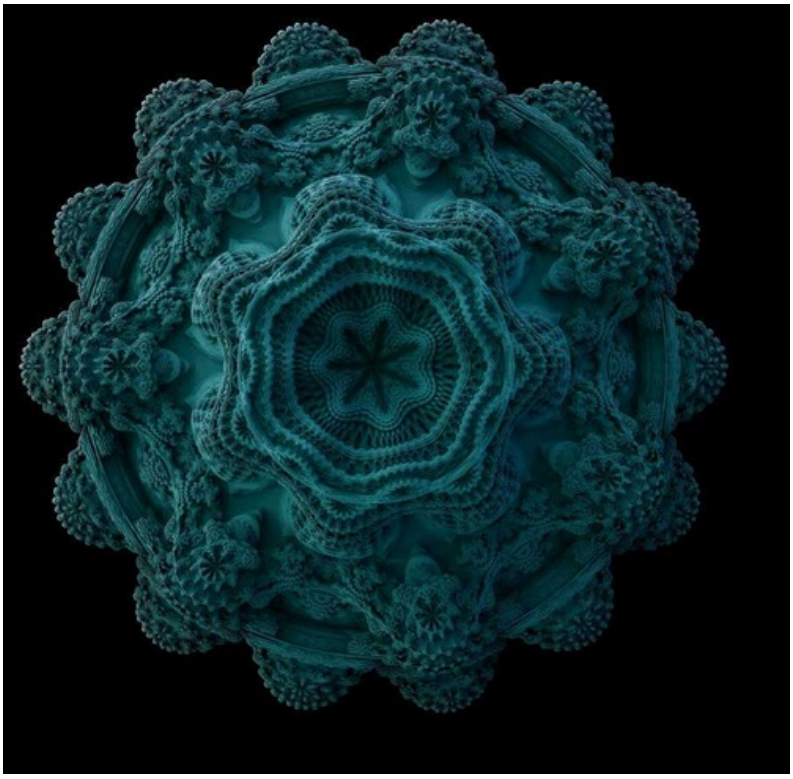
Algoritmo 1 – Pseudocódigo do Mandelbulb

```
1     r = sqrt(x*x + y*y + z*z )
2     theta = atan2(sqrt(x*x + y*y) / z)
3     phi = atan2(y/x)
4
5     novox = r^n * sin(theta*n) * cos(phi*n)
6     novoy = r^n * sin(theta*n) * sin(phi*n)
7     novoz = r^n * cos(theta*n)
```

Fonte: O autor (2022)

O valor do expoente n é a ordem do fractal de Mandelbulb. Para um valor de $n > 3$, o resultado é uma estrutura semelhante a um bulbo tridimensional, sendo que os detalhes de superfície e o número de lóbulos depende do valor de n. Na Figura 14 tem-se o fractal de Mandelbub utilizando um valor *n* igual a 8.

Figura 14 – Mandelbulb



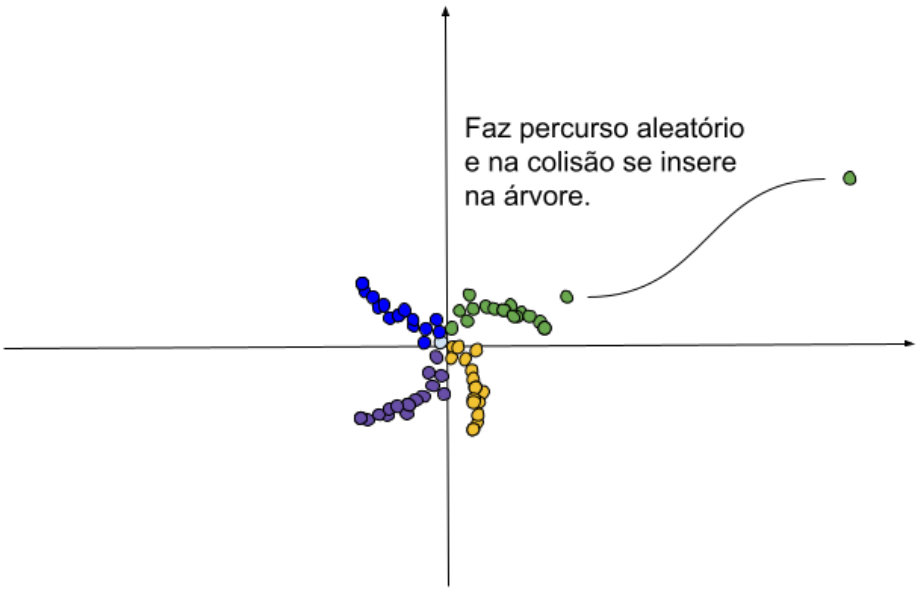
Fonte: Mori (2009).

2.2.3 Fractais Aleatórios

Os fractais aleatórios são criados a partir de métodos que possuem algum processo randômico. Em sua maioria, os fractais que representam formas da natureza se encaixam nessa categoria, sendo dois exemplos a árvore browniana e a montanha fractal.

A árvore browniana, também conhecida como agregação por difusão limitada, é obtida através de um ponto inicial, que por sua vez é obtido de forma aleatória. Após, todos os outros pontos que integram a árvore browniana são gerados fazendo uma curva aleatória sem direção até encostar em algum ponto que já está integrado na árvore. Uma vez que essa colisão acontece, o ponto será anexado ao ponto no qual ele encostou (BOURKE, 2014). Após, um novo ponto é gerado e esse processo é repetido. Na Figura 15 há um exemplo, onde um ponto gerado em uma posição aleatória realiza uma curva aleatória até ser anexado à árvore.

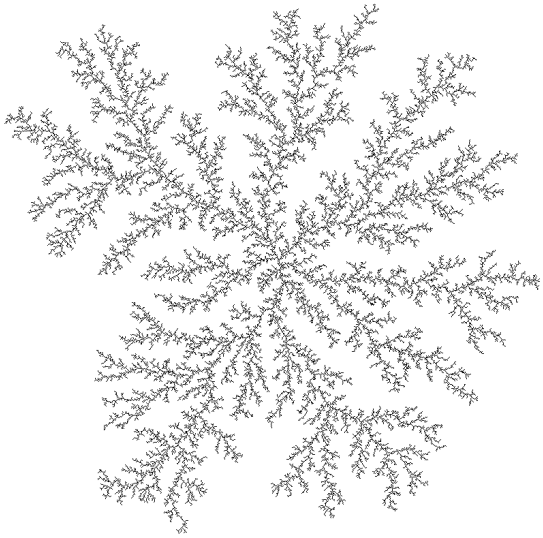
Figura 15 – Colisão e percurso de um ponto na árvore browniana



Fonte: O autor (2022).

Esse processo é executado de forma infinita ou até um limite definido, sendo que conforme o número de iterações aumenta, a quantidade de ramificações da árvore também aumenta, porém a semelhança sempre é mantida. A Figura 16 apresenta uma árvore browniana gerada com um número elevado de iterações.

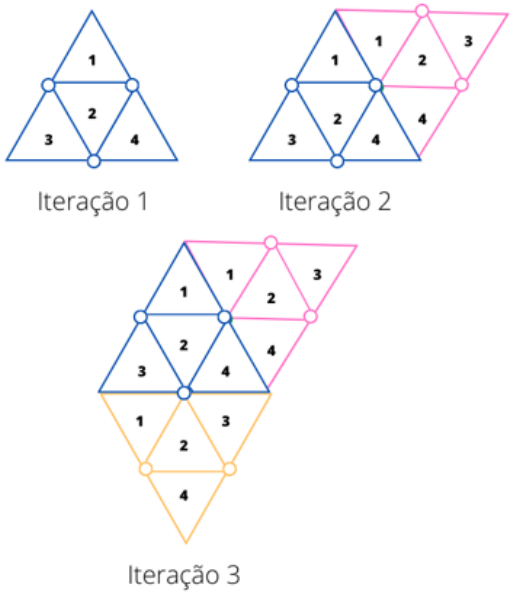
Figura 16 – Árvore browniana com número elevado de iterações



Fonte: Adaptado de Bourke (2014).

A montanha fractal é frequentemente utilizada no desenvolvimento de cenários para jogos. O desenvolvimento desse fractal tem início com três pontos, a partir dos quais é gerado um triângulo no espaço tridimensional. Acham-se os pontos centrais das 3 linhas que formam o triângulo e criam-se 4 novos triângulos a partir desse triângulo (Figura 17). Depois deslocam-se aleatoriamente esses pontos centrais para baixo ou para cima dentro de um intervalo de valores pré-estabelecido, criando-se um novo triângulo. Repete-se o mesmo procedimento, mas fazendo com que os limites do intervalo de valores sejam iguais à metade da iteração anterior.

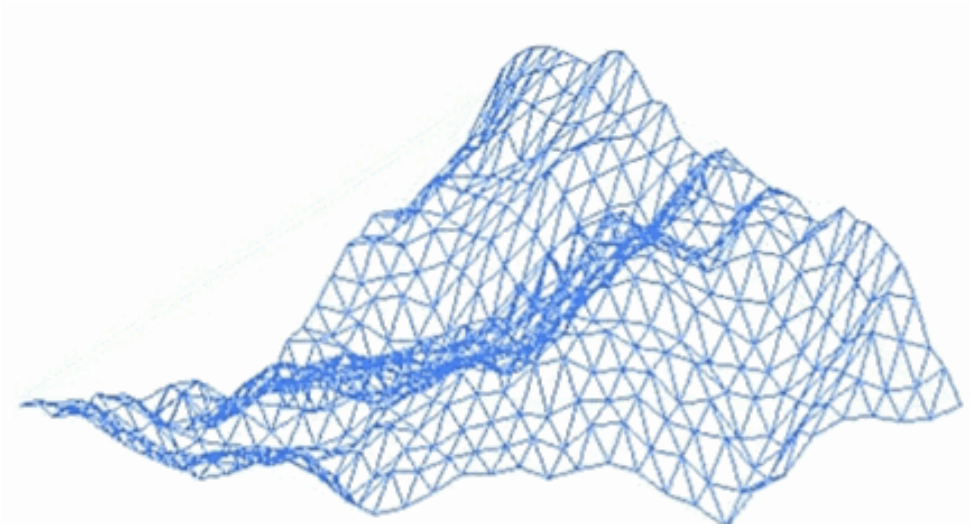
Figura 17 – Processo iterativo para geração de montanha fractal



Fonte: O autor (2022).

A montanha será formada a partir de diversas iterações até que o cenário resultante seja adequado. Após a geração da montanha, geralmente, é realizada a aplicação de uma textura. A Figura 18 mostra a formação de uma montanha fractal após algumas iterações.

Figura 18 – Montanha fractal



Fonte: Adaptado de Campos (2006).

3 UNIDADE DE PROCESSAMENTO GRÁFICO

Em 1965, Gordon Earle Moore, químico e cofundador da Intel, previu que a densidade dos transistores nas CPUs dobraria a cada 24 meses e, consequentemente, dobrar-se-ia o poder computacional no mesmo período (MOORE, 1965). Essa previsão é conhecida como a lei de Moore e até hoje é debatida na área da tecnologia. Em 2019, o CEO da NVIDIA, Jensen Huang, durante o evento da CES 2019 (*Consumer Electronics Show*), afirmou que a Lei de Moore não era mais válida, uma vez que o mercado tecnológico não estaria mais conseguindo colocá-la em prática (HUANG, 2019). Já o CEO da Intel, Pat Gelsinger, em uma entrevista em 2021, afirmou que a lei de Moore ainda é válida e que a empresa teria até mesmo uma previsão de ultrapassá-la (GELSINGER, 2021).

Independentemente do debate existente sobre a lei de Moore, existem dificuldades cada vez maiores para a produção de processadores mais rápidos. O aumento do número de transistores proporciona um aumento no poder computacional, entretanto tal crescimento introduz problemas físicos em relação à construção do processador. De fato, dificuldades como a dissipação de calor e a comunicação entre os transistores surgem com o acréscimo no número de transistores. Como uma alternativa para contornar essas dificuldades no desenvolvimento de processadores cada vez mais rápidos, foram criadas as arquiteturas paralelas, em que diversos processadores cooperam entre si para resolver um determinado problema (STALLINGS, 2017).

A computação paralela consiste na utilização de múltiplas unidades de processamento, trabalhando juntas para a execução de uma tarefa (CUNHA, 1997). Para que essas unidades possam trabalhar de forma cooperativa, é necessário que a tarefa seja dividida em partes, que podem ser executadas simultaneamente (SCHEPKE, 2018). Muitas vezes essas partes precisam utilizar algum mecanismo de comunicação e sincronização entre elas (SCHEPKE, 2018).

3.1 TAXONOMIA DE FLYNN

Existe uma grande variedade de arquiteturas paralelas, sendo que uma das formas mais utilizadas para a sua classificação é a Taxonomia de Flynn. Nesse caso, as arquiteturas são classificadas de acordo com o número de fluxos de instruções e do número fluxo de dados. A Figura 19 apresenta uma divisão das arquiteturas de acordo com a Taxonomia de Flynn.

Figura 19 – Taxonomia de Flynn

	SD (Single)	**MD (Multiple Data)**
SI (Single Instruction	SISD Máquinas von Neumann convencionais	SIMD Máquinas *Array*
MI (Multiple Instruction)	MISD Sem representante	MIMD Multiprocessadores e Multicomputadores

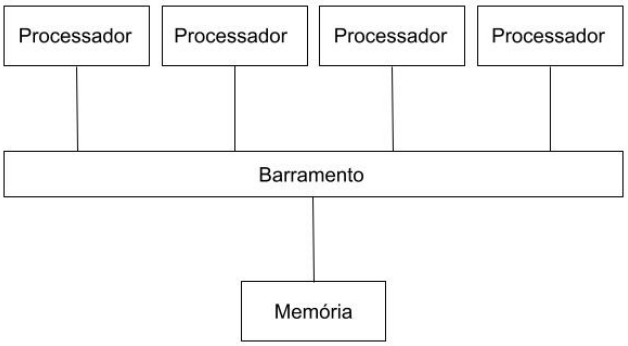
Fonte: Adaptado de Schepke (2018).

Como pode ser observado na Figura 19, a Taxonomia de Flynn abrange quatro classes de arquiteturas de computadores:

- SISD (*Single Instruction, Single Data*): os representantes desta categoria possuem apenas um fluxo de instruções e um único fluxo de dados. Os computadores que apresentam um único núcleo de processamento (*single core)* estão nessa categoria.
- SIMD (*Single Instruction, Multiple Data*): os representantes desta categoria apresentam apenas um fluxo de instruções, porém vários fluxos de dados, ou seja, uma única instrução é aplicada simultaneamente sobre um conjunto de dados. Os processadores vetoriais e matriciais pertencem a essa categoria. Mais recentemente, foram desenvolvidas as GPUs, que também pertencem a essa categoria.
- MISD (*Multiple Instruction, Single Data*): neste caso, várias instruções são executadas simultaneamente sobre um único fluxo de dados. Não existe nenhuma arquitetura representante desta categoria.
- MIMD (*Multiple Instruction, Multiple Data*): nesse tipo de arquitetura múltiplas instruções são executadas sobre diferentes fluxos de dados. Os computadores que possuem múltiplos núcleos de processamento estão nesta categoria, como por exemplo, os computadores *multicore*, os multiprocessadores e os *clusters* de computadores.

Os representantes da categoria MIMD podem ser divididos ainda de acordo com o compartilhamento da memória em: multiprocessadores (memória compartilhada) e multicomputadores (memória distribuída). Os multiprocessadores são formados por múltiplas unidades de processamento que possuem acesso a uma única área de memória compartilhada. A sincronização e a comunicação entre as tarefas são realizadas através dessa área de memória compartilhada. Essa arquitetura apresenta um menor custo de comunicação, uma vez que a comunicação é realizada através de um barramento. Entretanto, o *hardware* apresenta um custo mais elevado e um limite físico em relação ao número de processadores. De fato, essas arquiteturas apresentam uma contenção no acesso ao barramento, sendo que essa disputa aumenta com o crescimento do número de processadores (PARHAMI, 2002). A Figura 20 representa a arquitetura de um multiprocessador.

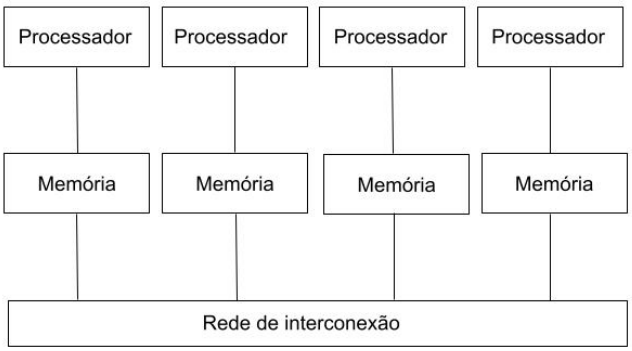
Figura 20 – Multiprocessador



Fonte: O autor (2022).

Nos multicomputadores, cada unidade de processamento possui sua própria memória, que não pode ser acessada de forma direta pelas outras unidades de processamento. Dessa forma, a sincronização e a comunicação entre as tarefas são realizadas através da troca de mensagens entre elas. Essa comunicação é feita por meio de uma rede de comunicação, possuindo um elevado *overhead* e inviabilizando a utilização desse tipo de arquitetura em determinadas aplicações (CORSO, 1999). Contudo, os multicomputadores apresentam um menor custo de fabricação e não possuem um limite teórico no número de processadores. Na Figura 21 tem-se um exemplo da arquitetura de um multicomputador.

Figura 21 – Multicomputador



Fonte: O autor (2022).

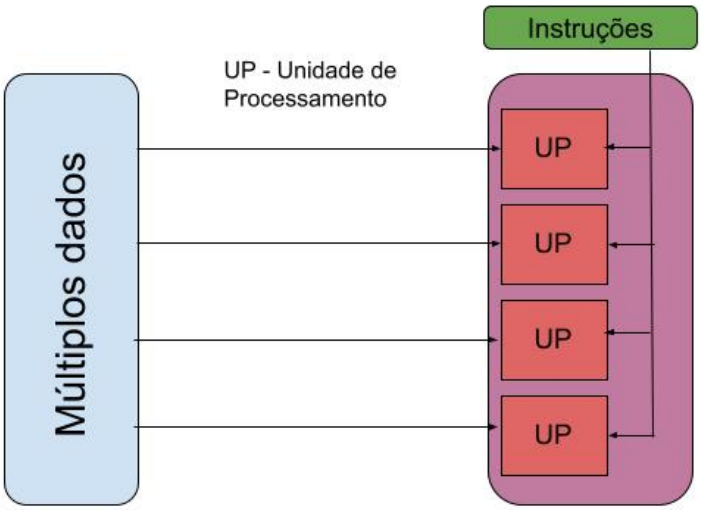
No presente Trabalho de Conclusão de Curso, a arquitetura utilizada para a paralelização das aplicações de fractais são as GPUs, que se encontram na categoria SIMD. Sendo assim, nas próximas seções, a arquitetura SIMD será descrita com maiores detalhes. Uma ênfase maior foi dada para a apresentação da arquitetura das GPUs da NVIDIA, uma vez que essa foi a arquitetura utilizada no desenvolvimento deste trabalho.

3.2 *SINGLE INSTRUCTION, MULTIPLE DATA*

Os princípios da arquitetura SIMD são datados da década de 60, quando essa arquitetura surgiu como uma alternativa para a execução de aplicações científicas e comerciais que executavam uma grande quantidade de cálculos e/ou manipulavam um grande volume de dados. O primeiro representante dessa arquitetura foi o supercomputador ILLIAC IV, desenvolvido pela Universidade de Illinois em 1965 e que, em seu primeiro projeto, possuía 256 processadores (PARHAMI, 2002). No entanto, o representante com maior êxito desta categoria foi o supercomputador *Cray 1*.

A arquitetura SIMD consiste em uma única unidade de controle que gerencia vários núcleos de processamento. Essa unidade de controle recebe um fluxo de instruções, sendo responsável pela distribuição das mesmas entre os núcleos de processamento disponíveis, que executam essas instruções de forma simultânea. A comunicação entre as unidades de processamento é realizada através de uma área de memória compartilhada. Na Figura 22 há uma representação da arquitetura SIMD.

Figura 22 – Arquitetura SIMD



Fonte: O autor (2022).

As arquiteturas SIMD podem ser divididas em duas categorias: os processadores matriciais e os processadores vetoriais (BASU, 2016). Um processador vetorial é um processador de uso geral que pode efetuar operações sobre valores escalares. Já um processador matricial não possui tais funções e depende de uma unidade de controle externa para executar operações escalares (LEMES, 2020).

3.3 UNIDADES DE PROCESSAMENTO GRÁFICO

Até a década de 90 os supercomputadores eram os únicos representantes da arquitetura SIMD. No início da década de 90, porém, surgiram as primeiras unidades de processamento gráfico (GPUs, do inglês *Graphics Processing Unit*). Inicialmente, as placas gráficas eram utilizadas para aceleração gráfica 2D e 3D (SANDERS; KANDROT, 2010). No entanto, atualmente elas são utilizadas para a execução de aplicações de uso geral (SETH, 2021).

Uma CPU (do inglês *central processing unit*) é formada por um número pequeno de núcleos, com um sofisticado controle lógico e com a presença de memórias *caches* de tamanho considerável. Já uma GPU é formada por uma quantidade muito maior de núcleos, porém mais simples. Por exemplo, o processador *Intel® Core™ i9-9900K* possui 8 núcleos, enquanto uma placa gráfica *GeForce RTX 3090*, da empresa NVIDIA, possui 10.496 núcleos de processamento.

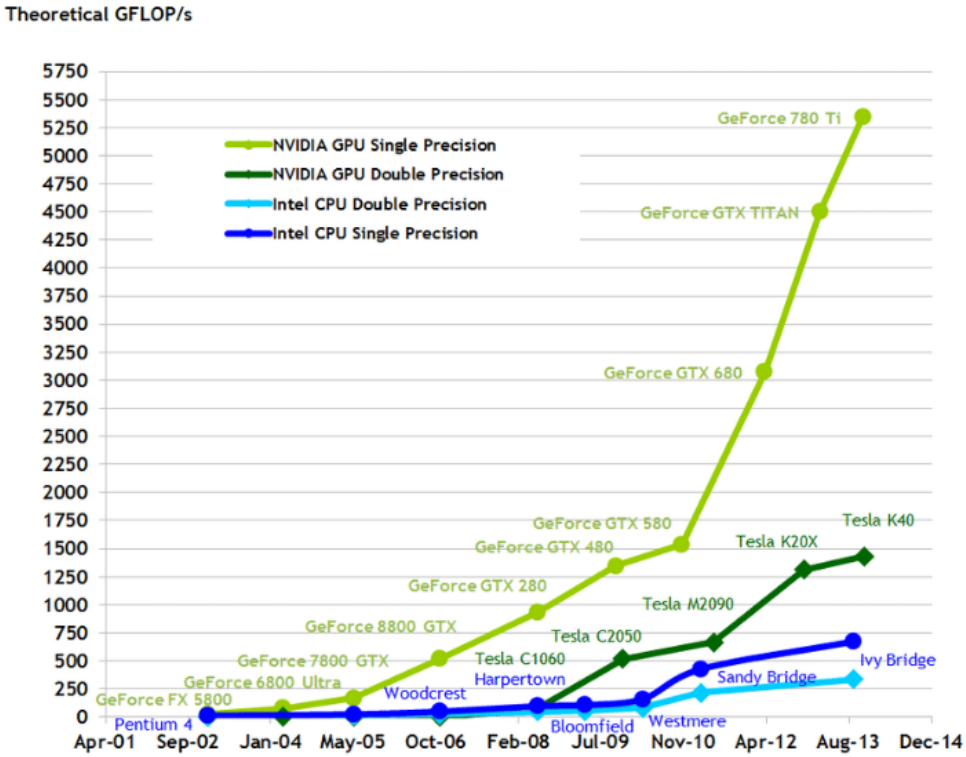
A Figura 23 apresenta uma comparação entre a arquitetura de uma CPU e de uma GPU. Como pode ser observado, a CPU possui uma grande quantidade de circuitos dedicados às unidades de controle, memórias *caches* e um número limitado de unidades de processamento. Já a GPU tem áreas menores para a memória *cache* e unidade de controle, possuindo um grande número de unidades de processamento e, consequentemente, uma maior capacidade de processamento (TSE, 2012).

Figura 23 – Arquitetura de uma CPU vs GPU

Fonte: Tse (2012).

Essa diferença no número de unidades de processamento é o principal motivo para uma GPU apresentar um poder computacional superior quando comparada com uma CPU. Na Figura 24 é apresentado um comparativo entre o número de operações em ponto flutuante que são executadas por uma GPU e por uma CPU. Como pode ser observado, essa diferença vem crescendo significativamente com o passar dos anos e com a evolução das GPUs.

Figura 24 – Comparação de operações de ponto flutuante entre GPU vs CPU



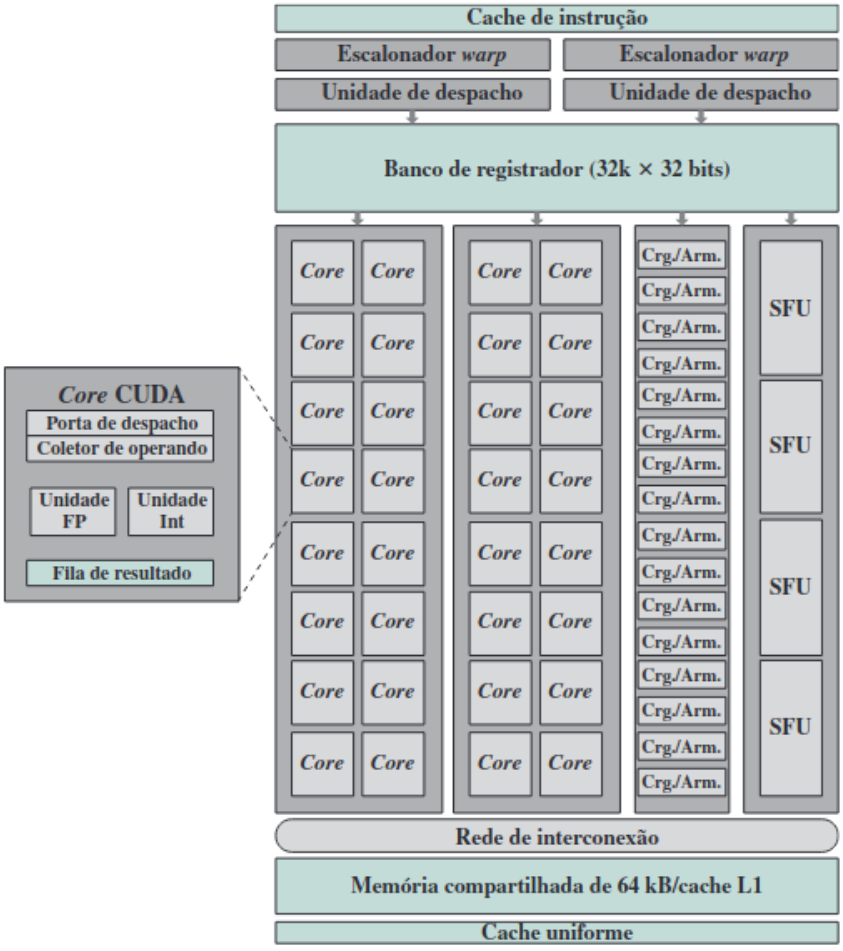
Fonte: Závodszky (2015).

3.4 ARQUITETURA DA GPU

Pesquisas realizadas mostram que as placas de vídeo mais vendidas atualmente são fabricadas pela empresa NVIDIA (SHILOV, 2021). Inclusive, essa é a fabricante da GPU que foi utilizada para o desenvolvimento deste trabalho. Dessa forma, nessa seção será realizada uma breve descrição da arquitetura das GPUs da empresa NVIDIA.

Uma GPU da NVIDIA é composta pela união de vários blocos de unidades de processamento que são chamados de SMs (*Streaming Multiprocessors*). Esses blocos são a base de construção da GPU, sendo responsáveis pela execução de um grupo de *threads*[1], chamados de *warps* (ZANOTTO; FERREIRA; MATSUMOTO, 2019). Cada SM é formado por um conjunto de núcleos de processamento chamados de CUDA *cores*, que compartilham de uma memória *cache* de nível 1 (*cache* L1). A estrutura de um CUDA *core* é composta por dois *pipelines* [2], um para a execução de operações de ponto flutuante e outro para operações com números inteiros, sendo que somente um dos *pipelines* pode ser utilizado por ciclo do relógio. Por fim, cada SM possui 4 SFUs, os quais são utilizados para a execução de operações mais complexas, como operações de seno, cosseno e raiz quadrada. A Figura 25 ilustra a arquitetura de um único SM, que possui 32 CUDA *cores* e uma memória *cache* de nível 1 de $64Kb$.

Figura 25 – Arquitetura de um SM



Fonte: Stallings (2017).

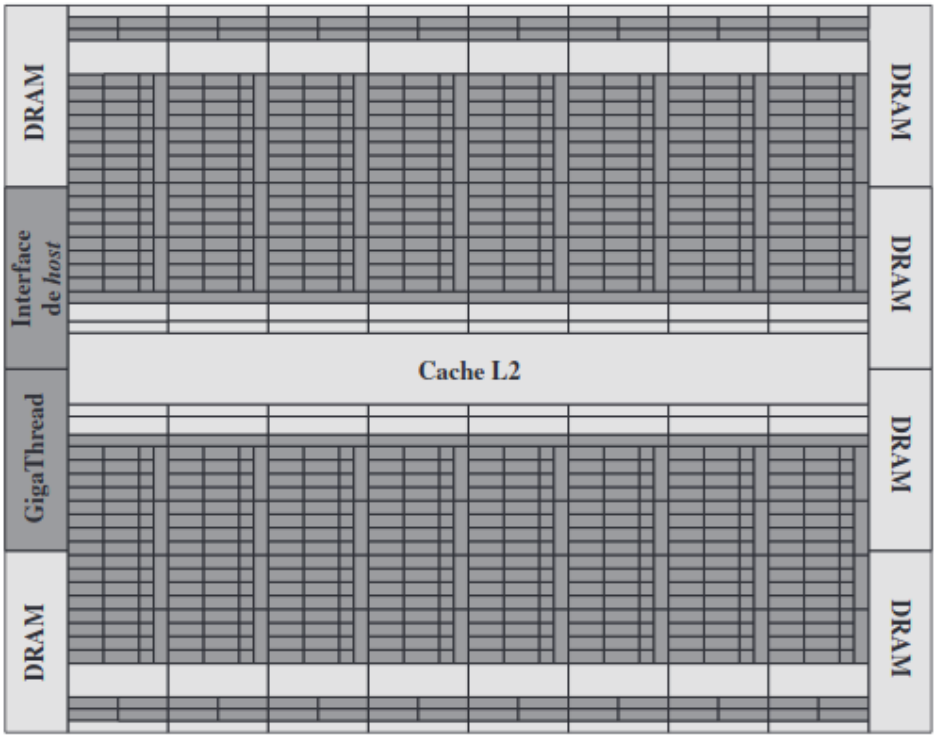
[1] Fluxo de execução dentro de um processo, ou seja, as sequências de instruções a serem executadas dentro de um programa

[2] Trata-se de uma técnica para executar o paralelismo em nível de instrução em um único processador. Em vez de o processador executar toda a instrução de uma vez, ela é dividida em etapas. Assim que a primeira fase termina, a segunda etapa é executada e uma nova instrução é executada na primeira fase. Desta forma, a CPU está quase sempre ocupada, reduzindo o tempo de processamento das instruções.

Uma GPU possui um escalonador global chamado de *GigaThread* (Figura 26), que é responsável por distribuir os *warps* de *threads* entre os SMs disponíveis. Esse escalonador separa cada bloco de *thread* em agrupamentos de 32 *threads*, que são distribuídos entre os SMs[3]. Desta forma, um *warp* consiste basicamente em um conjunto de *threads* que executam simultaneamente uma mesma instrução, sendo que essa execução é controlada por uma única unidade de controle. Em caso de desvios em um *warp*, esses são executados de forma serial e no final convergem para o caminho original da execução (LINDHOLM *et al.*, 2008).

Como já mencionado, os SMs possuem um conjunto próprio de registradores e uma memória *cache* própria. Além disso, a GPU possui ainda uma memória global (*cache* L2) que é compartilhada por todos os núcleos de todos os SMs. No entanto, o uso dessa memória deve ser minimizado, uma vez que essa se localiza fora do SM e apresenta uma alta latência de acesso. A Figura 26 apresenta a arquitetura Fermi da NVIDIA, que possui 16 SMs, cada um contendo 32 CUDA *cores*, totalizando 512 núcleos.

Figura 26 – Arquitetura Fermi da NVIDIA



Fonte: Stallings (2017).

[3] Nas arquiteturas das GPUs NVIDIA, cada SM possui 32 CUDA *cores*

3.4.1 Hierarquia de Memória de uma GPU

Uma GPU possui diferentes tipos de memória que diferem no tempo de acesso, localização e escopo. De forma geral, os níveis de memória de uma GPU são:

- Registradores: é o tipo de memória com maior velocidade de acesso, visto que se encontram próximos aos núcleos de processamento. Cada SM tem o seu próprio conjunto de registradores, sendo que as *threads* de um *warp* podem acessar somente os seus próprios registradores.
- Memória compartilhada: cada SMs possui sua própria memória compartilhada, a qual pode ser acessada por todas as *threads* de um *warp*. Assim, essa área de memória pode ser utilizada para a comunicação das *threads* de um *warp*. A utilização da memória compartilhada diminui o acesso à memória global, reduzindo assim a latência de comunicação (KINDRATENKO, 2014).
- Memórias locais: são utilizadas como memórias secundárias aos registradores, portanto só podem ser acessadas pelas *threads* de um *warp*. Elas são utilizadas, geralmente, para alocar variáveis locais com tamanho maior que os registradores.
- Memória global (DRAM): encontra-se localizada fora do *chip* e apresenta uma baixa velocidade de acesso. No entanto, essa é a memória que possui a maior capacidade de armazenamento dentro da hierarquia de memória de uma GPU. Essa memória possibilita a comunicação entre todas as *threads* de todos os SMs, podendo ser acessada também pela CPU.
- Memória constante: essa é uma memória com baixa latência e uma alta velocidade, funcionando apenas no modo de leitura (*read-only*). A memória constante pode ser acessada por todas as *threads* de todos os SMs. Essa memória apresenta um tamanho pequeno e se localiza junto à memória global, podendo ser utilizada como uma memória *cache*.
- Memória de textura: esse tipo de memória divide o espaço físico com a memória global, podendo ser acessada por todas as *threads* de todos os SMs. Essa é uma memória que opera somente no modo de leitura. Ela possui uma velocidade de acesso idêntica à memória global, sendo otimizada para o acesso a dados organizados no formato 1D, 2D ou 3D.

No Quadro 1 tem-se um resumo dos tipos de memórias que se encontram disponíveis na hierarquia de memória de uma GPU.

Quadro 1 – Tipos de memória em uma GPU

Memória	**Tempo de Acesso**	**Tipo de Acesso**	**Escopo**	**Vida útil dos dados**
Registradores	O mais rápido. *On-chip*	W/R	*Thread* única	*Thread*
Compartilhada	Rápido. *On-chip*	W/R	Todas as threads em um bloco	Bloco
Local	Entre 100 e 15 vezes mais lento que a memória compartilhada e os registradores. *Off-chip*	W/R	*Thread* única	*Thread*
Global	Entre 100 e 15 vezes mais lento que a memória compartilhada e os registradores. *Off-chip*	W/R	Todas as *threads* e *host*	Aplicação
Constante	Entre 100 e 15 vezes mais lento que a memória compartilhada e os registradores. *Off-chip*	R	Todas as *threads* e *host*	Aplicação
Textura	Entre 100 e 15 vezes mais lento que a memória compartilhada e os registradores. *Off-chip*	R	Todas as *threads* e *host*	Aplicação

Fonte: Adaptado de (STALLINGS, 2017).

3.4.2 *Compute Unified Device Architecture* (CUDA)

Nos últimos anos foram criadas diversas plataformas e bibliotecas que facilitam o desenvolvimento de aplicações para GPUs. Entre elas, a biblioteca que apresenta um maior destaque é a CUDA (*Compute Unified Device Architeture*). A CUDA é uma plataforma de desenvolvimento criada pela NVIDIA e que facilita o desenvolvimento de aplicações paralelas para serem executadas especificamente nas GPUs da empresa. A CUDA possibilita ao programador abstrair as complexidades do *hardware*, evitando que o mesmo necessite aprender um novo conjunto de instruções a cada aprimoramento das GPUs (TSE, 2012).

A programação em CUDA é considerada heterogênea, uma vez que envolve a execução de um programa em duas plataformas diferentes: o *host* e o *device*. O *host* corresponde ao computador onde a GPU encontra-se instalada e o *device* consiste na placa gráfica que possui os núcleos CUDAs. Assim, o código fonte de um programa CUDA é uma mistura de códigos em um mesmo arquivo, em que parte do código é executado no *host* e a outra parte no *device* (TSE, 2012).

Durante o processo de compilação, o compilador NVIDIA C *Compiler* (NVCC) cria dois arquivos, sendo que o primeiro contém o código do *host* e o segundo o código para o *device*. A separação do código para o *device* e para o *host* é realizada pelo NVCC através do uso de palavras-chave, onde as funções e métodos que foram desenvolvidas para serem executadas no *device* são chamadas de *kernels*. Uma função *kernel* é definida pela utilização da palavra-chave *global*. No Algoritmo 2 há a declaração de uma função *kernel* que realiza a soma de duas matrizes.

Algoritmo 2 – Exemplo de código executado no *device*

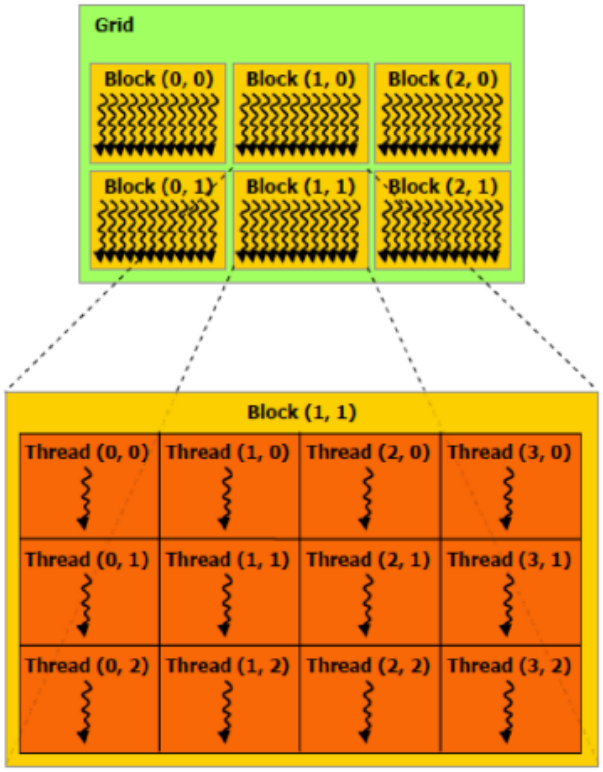
```
1 // Kernel Definition
2 __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
3     int i = blockIdx.x * blockDim.x + threadIdx.x;
4     int j = blockIdx.y * blockDim.y + threadIdx.y;
5
6     if (i < N && j < N) {
7         C[i][j] = A[i][j] + B[i][j];
8     }
9 }
```

Fonte: Tse (2012)

O código desenvolvido para o *host* pode ser compilado utilizando um compilador C/C++ padrão, como o compilador gcc. Já o código para o *device* é compilado utilizando o compilador CUDA C *Compiler* (CUDACC), que gera código objeto em uma linguagem *Assembly* conhecida como *Parallel Thread eXecution* (arquivos PTX). Os arquivos PTX são reconhecidos pelos *drivers* instalados nas placas gráficas da NVIDIA. Por fim, o código resultante é agrupado gerando um único executável (TSE, 2012).

A execução de um programa CUDA inicia no *host*, onde são inicializadas as *threads* que serão executadas no *device*. O número de *threads* é definido pelo programador, sendo informado nas funções *kernel*. O conjunto de *threads* que será responsável pela execução de uma função *kernel* é chamado de *grid* (TSE, 2012), sendo que um *grid* pode ser composto de um ou mais blocos de *threads*. A Figura 27 mostra a organização de um *grid*, que apresenta uma dimensão 2×3 com blocos de *threads* que têm uma dimensão de 3×4. Quando um *kernel* é executado, o *grid* e os blocos de *threads* são criados. Os blocos são designados a um SM, sendo que cada SM pode executar os blocos de forma concorrente. Os blocos restantes são colocados em uma fila até que um SM esteja livre. A quantidade de blocos varia de GPU para GPU.

Figura 27 – *Grid* de blocos de *threads*



Fonte: Nvidia (2022).

O Algoritmo 3 apresenta a inicialização das dimensões de um *grid* e de um bloco de *threads* em CUDA. Na linha 6 do Algoritmo 3 é definida a dimensão do *grid* e na linha 5 são definidas as dimensões de cada um dos blocos. Como pode ser observado, foram definidos blocos de 16×16 *threads*. As *threads* de um mesmo bloco podem se comunicar entre si através da memória compartilhada. A sincronização das *threads* de um mesmo bloco pode ser feita através da função *syncthreads* (linha 8). Essa função consiste em uma barreira, sendo que todas as *threads* de um mesmo bloco esperam a execução das outras antes de prosseguir.

Algoritmo 3 – Código com definição de threads, blocos e grid

```
1  #define BLOCK_SIZE 16
2
3  dim3 threadsPerBlock(16, 16);
4  dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
5  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
6  dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
7
8  __syncthreads();
```

Fonte: Tse (2012)

As áreas de memória do *host* e do *device* são separadas, ou seja, as variáveis e os dados presentes na memória do *host* não podem ser acessadas pela GPU. Dessa forma, torna-se necessário realizar a transferência dos dados da memória do *host* para a memória da GPU. Isso pode ser executado através das funções *cudaMalloc* e *cudaMemcpy*. O Algoritmo 4 exemplifica a alocação de vetores na memória do *device* através da utilização da função *cudaMalloc*, que é utilizada nas linhas 11, 13 e 15. Já a transferência dos vetores para a memória do *device* é realizada por meio da função *cudaMemcpy*, que pode ser observada nas linhas 18 e 19.

Algoritmo 4 – Alocação de memória no *device*

```
1   // Allocate input vectors h_A and h_B in host memory
2   float* h_A = (float*)malloc(size);
3   float* h_B = (float*)malloc(size);
4   float* h_C = (float*)malloc(size);
5
6   // Initialize input vectors
7   ...
8
9   // Allocate vectors in device memory
10  float* d_A;
11  cudaMalloc(&d_A, size);
12  float* d_B;
13  cudaMalloc(&d_B, size);
14  float* d_C;
15  cudaMalloc(&d_C, size);
16
17  // Copy vectors from host memory to device memory
18  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
19  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Fonte: Nvidia (2022)

A memória global da GPU pode ser acessada pela palavra-chave *device* e variáveis alocadas nela podem ser acessadas durante a execução de toda a aplicação. A memória constante é identificada através da palavra-chave *constant*, sendo que os dados ali armazenados podem ser acessados durante a execução de toda a aplicação. Já a memória compartilhada pode ser acessada através da palavra-chave *shared*, sendo que as variáveis podem ser acessadas somente durante a execução do *kernel*. Por fim, os registradores são variáveis utilizadas dentro de um *kernel* e são alocadas automaticamente durante a execução. No algoritmo 5 há um exemplo da utilização das palavras-chave para o acesso às diferentes áreas de memória.

Algoritmo 5 – Acesso aos tipos de memória

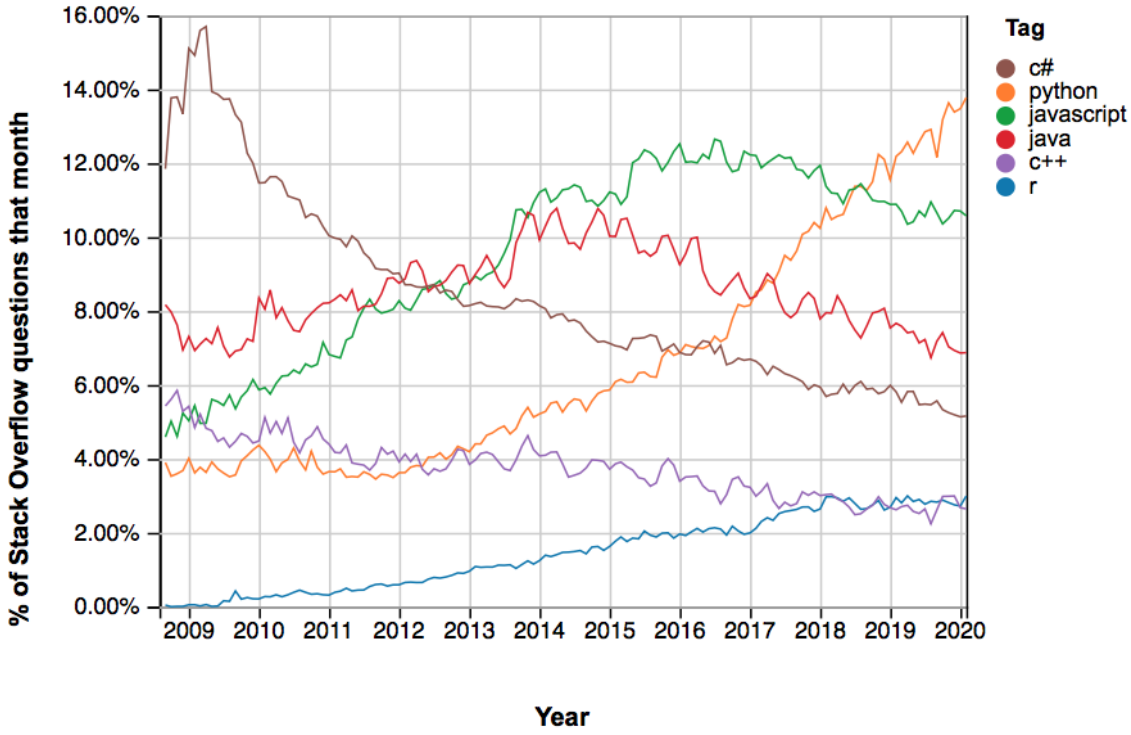
```
1   __constant__ float constData[256];
2   float data[256];
3   cudaMemcpyToSymbol(constData, data, sizeof(data));
4   cudaMemcpyFromSymbol(data, constData, sizeof(data));
5
6   __shared__ float devData;
7   float value = 3.14f;
8   cudaMemcpyToSymbol(devData, &value, sizeof(float));
9
10  __device__ float* devPointer;
11  float* ptr;
12  cudaMalloc(&ptr, 256 * sizeof(float));
13  cudaMemcpyToSymbol(devPointer, &ptr, sizeof(ptr));
```

Fonte: Nvidia (2022)

4 IMPLEMENTAÇÃO SEQUENCIAL

As implementações foram desenvolvidas utilizando a linguagem de programação Python, visto que essa é uma das linguagens mais utilizadas atualmente, devido, principalmente, à facilidade de aprendizado e ao grande número de pacotes disponíveis. O crescimento da utilização da linguagem Python pode ser ilustrado através do gráfico da Figura 28, que apresenta o aumento no número de perguntas sobre essa linguagem no fórum *StackOverflow*.

Figura 28 – Crescimento de utilização da linguagem de programação Python



Fonte: Joury (2020).

No que se refere à visualização de fractais, um grande número de pacotes estão disponíveis, como a biblioteca Matplotlib (MATPLOTLIB, 2022), a biblioteca Pillow (PILLOW, 2022) e a biblioteca RayPy (RAYPY, 2022). Neste trabalho, optou-se pela utilização da biblioteca Matplotlib em decorrência da facilidade de integração da mesma com a biblioteca NumPy, que foi a biblioteca utilizada para o armazenamento e geração dos fractais.

4.1 DEFINIÇÃO DOS FRACTAIS

A área de fractais é amplamente utilizada para a realização de testes sobre ambientes e ferramentas de programação paralela. Por exemplo, no trabalho *An overview of parallel visualisation methods for Mandelbrot and Julia sets* (DRAKOPOULOS; MIMIKOU; THEOHARIS, 2003) foram desenvolvidas implementações paralelas dos fractais de Mandelbrot e Julia para *clusters* de computadores. Para isso, foi utilizada a biblioteca de troca de mensagens MPI (*Message Passing Interface*) (PACHECO; MALENSEK, 2021), bem como a biblioteca de desenvolvimento MPE (*MPI Parallel Environment*), que é uma biblioteca para a depuração de programas MPI (GROPP *et al.*, 1995).

No trabalho *Parallel Implementation and Analysis of Mandelbrot Set Construction* (GäNG *et al.*, 2008) e no trabalho *Processamento Paralelo na Resolução do Fractal de Mandelbrot* (SANTOS; DOMINGUEZ, 2012) são apresentadas implementações paralelas do fractal de Mandelbrot também utilizando a biblioteca MPI. Já no trabalho *MPI vs OpenMP: A case study on parallel generation of Mandelbrot set* é realizado um estudo comparativo entre arquiteturas de memória distribuída e compartilhada utilizando o fractal de Mandelbrot (TRACOLLI, 2016). Para o desenvolvimento foram utilizadas as bibliotecas de troca de mensagens MPI e a biblioteca de *threads* OpenMP (CHAPMAN; JOST; PAS, 2007).

No trabalho *Implementation and Analysis of Fractals Shapes using GPU-CUDA Model* foram desenvolvidas versões paralelas dos fractais de Julia e Mandelbrot para arquiteturas de GPU utilizando a biblioteca CUDA (SALLOW, 2021). No trabalho *Paradigmas de Processamento Paralelo na Resolução do Fractal de Mandelbrot* (SANTOS; DOMINGUEZ, 2012) e no trabalho *O Problema do Fractal de Mandelbrot como Comparativo de Arquiteturas de Memória Compartilhada–GPU vs OpenMP* (SANTOS; DOMINGUEZ; ORELLANA, 2011) são realizados estudos comparativos entre implementações do fractal de Mandelbrot para GPU, utilizando CUDA, e ambientes de memória compartilhada, utilizando OpenMP.

Já no trabalho *Benchmark comparison of computing the Mandelbrot set in OpenCL* (HUSEINOVIć; RIBIć, 2015) foi realizado um estudo sobre a paralelização do fractal de Mandelbrot, utilizando a biblioteca OpenCL. Essa é uma biblioteca para o desenvolvimento de programas para plataformas heterogêneas, consistindo em CPUs e GPUs. Por fim, no trabalho *Performance comparison in simulation of Mandelbrot set fractals using Numba* (HUNGILO; EMMANUEL; PRANOWO, 2020) é realizado um estudo sobre a paralelização do fractal de Mandelbrot em Python, utilizando Numba.

No Quadro 2 é realizada uma sistematização dos trabalhos citados. Como pode ser observado, os fractais de Mandelbrot e Julia são os mais utilizados em estudos que tratam de fractais gerados computacionalmente. Dessa forma, inicialmente optou-se pela implementação e paralelização de ambos os fractais. Além disso, optou-se pela implementação e paralelização do fractal de Mandelbrot 3D (conhecido como Mandelbulb), que apresenta uma implementação paralela em CUDA (MOMMERSTEEG, 2014). No entanto, não foram encontradas referências de implementações paralelas realizadas em Numba.

Quadro 2 – Trabalhos com fractais

Trabalho	**Fractal**	**Arquitetura**
(DRAKOPOULOS; MIMIKOU; THEOHARIS, 2003)	Julia e Mandelbrot	MPI
(SALLOW, 2021)	Julia e Mandelbrot	Cuda
(GäNG *et al.*, 2008)	Mandelbrot	MPI
(SANTOS; DOMINGUEZ; ORELLANA, 2011)	Mandelbrot	OpenMP e CUDA
(SANTOS; DOMINGUEZ, 2012)	Mandelbrot	OpenMP e CUDA
(HUSEINOVIć; RIBIć, 2015)	Mandelbrot	OpenCL
(TRACOLLI, 2016)	Mandelbrot	MPI
(GÓMEZ, 2020)	Mandelbrot	MPI e OpenMP
(HUNGILO; EMMANUEL; PRANOWO, 2020)	Mandelbrot	Numba
(MOMMERSTEEG, 2014)	Mandelbulb	CUDA
(XU, 2014)	Árvore browniana (DLA)	CUDA

Fonte: O Autor (2022).

A maioria dos estudos abordam fractais que são gerados através de relações de recorrência, em que o cálculo do fractal em cada ponto do espaço é independente e, consequentemente, são de fácil paralelização. Sendo assim, optou-se pela implementação e paralelização de um fractal aleatório. O fractal escolhido foi o fractal da árvore browniana, cuja paralelização foi abordada no trabalho *Performance Optimization for DLA Model Based on GPU* (XU, 2014), o qual apresenta uma implementação paralela em CUDA da árvore browniana.

4.1.1 Implementação sequencial dos fractais de Mandelbrot e Julia

As implementações dos fractais de Mandelbrot e Julia foram realizadas utilizando as bibliotecas NumPy e Matplotlib. A biblioteca NumPy é uma biblioteca desenvolvida para trabalhar com computação científica, possuindo tipos de dados necessários para a manipulação de vetores e matrizes (NUMPY, 2022). Ela foi utilizada para o cálculo e armazenamento dos fractais gerados. Já a biblioteca Matplotlib possui funções para facilitar a renderização e manipulação de imagens e foi utilizada para a visualização dos fractais.

No Algoritmo 6 é apresentado o programa sequencial utilizado para gerar o fractal de Mandelbrot. Ele é uma adaptação do código disponível em NUMBA (2022). Na linha 10 é possível observar a aplicação da equação de recorrência (Equação 2.4), utilizada para calcular cada ponto do fractal. Como pode ser observado na linha 28, os pontos do fractal são armazenados em um *array* da biblioteca NumPy, mais especificamente um *array* bidimensional, inicializado com zeros. Cada posição desse *array* vai receber um valor entre 0 a 255, resultante do cálculo dos fractais, que indica a cor do ponto. A visualização desses pontos é realizada utilizando a biblioteca Matplotlib (linha 34).

Algoritmo 6 – Código do Fractal de Mandelbrot

```
1  import numpy as np
2  from matplotlib import pyplot as plt
3  from pylab import imshow, show
4  from timeit import default_timer as timer
5
6  def mandelbrot(x, y, iteracoes):
7      c = complex(x, y)
8      z = 0.0j
9      for i in range(iteracoes):
10         z = z * z + c
11         if (z.real * z.real + z.imag * z.imag) >= 4:
12             return i
13     return iteracoes
14
15 def criar_fractal(min_x, max_x, min_y, max_y, imagem, iteracoes):
16     altura = imagem.shape[0]
17     largura = imagem.shape[1]
18     tamanho_pixel_x = (max_x - min_x) / largura
19     tamanho_pixel_y = (max_y - min_y) / altura
20
21     for x in range(largura):
22         real = min_x + x * tamanho_pixel_x
23         for y in range(altura):
24             imag = min_y + y * tamanho_pixel_y
25             cor = mandelbrot(real, imag, iteracoes)
26             imagem[y, x] = cor
27
28 imagem = np.zeros((4000, 4000), dtype=np.uint8)
29 inicio = timer()
30 criar_fractal(-2.0, 1.0, -1.0, 1.0, imagem, 1000)
31 tempo_exec = timer() - inicio
32 print(f"Tempo:{tempo_exec}")
33 imshow(imagem)
34 show()
```

Fonte: Adaptado de Numba (2022b)

No Algoritmo 7 é apresentada a implementação do fractal de Julia. Como pode ser observado, esse fractal utiliza a mesma equação de recorrência do fractal de Mandelbrot (linha 10). Porém, ele possui um valor diferente para a constante c, como pode ser visto na linha 7.

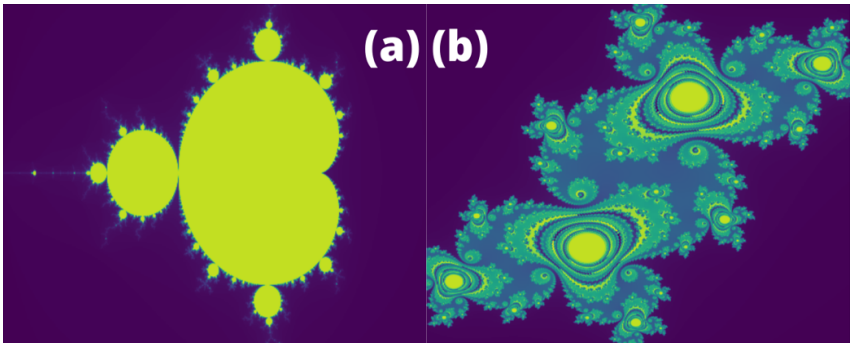
Algoritmo 7 – Julia em Python

```
1  import numpy as np
2  from matplotlib import pyplot as plt
3  from pylab import imshow, show
4  from timeit import default_timer as timer
5
6  def julia(x, y, iteracoes):
7      c = complex(-0.1, 0.65)
8      z = complex(x, y)
9      for i in range(iteracoes):
10         z = z * z + c
11         if (z.real * z.real + z.imag * z.imag) >= 4:
12             return i
13     return iteracoes
14
15 def criar_fractal(min_x, max_x, min_y, max_y, imagem, iteracoes):
16     altura = imagem.shape[0]
17     largura = imagem.shape[1]
18     tamanho_pixel_x = (max_x - min_x) / largura
19     tamanho_pixel_y = (max_y - min_y) / altura
20
21     for x in range(largura):
22         real = min_x + x * tamanho_pixel_x
23         for y in range(altura):
24             imag = min_y + y * tamanho_pixel_y
25             cor = julia(real, imag, iteracoes)
26             imagem[y, x] = cor
27
28 imagem = np.zeros((4000, 4000), dtype=np.uint8)
29 inicio = timer()
30 criar_fractal(-1.0, 1.0, -1.0, 1.0, imagem, 1000)
31 tempo_exec = timer() - inicio
32 print(f"Tempo:{tempo_exec}")
33 imshow(imagem)
34 show()
```

Fonte: O autor (2022)

Na Figura 29 tem-se os gráficos gerados a partir dos fractais implementados. Na Figura 29 (a) há o resultado do fractal de Mandelbrot e na Figura 29 (b) há o resultado do fractal de Julia, utilizando uma constante c igual a $-0.1 + 0.65i$. Ambos os fractais foram gerados utilizando uma resolução de 4.000×4.000 pontos. O fractal de Mandelbrot foi gerado utilizando valores x no intervalo de -2.0 a 1.0 e valores de y no intervalo de -1.0 a 1.0. Já o fractal de Julia foi gerado utilizando valores de x e de y nos intervalos de -1.0 a 1.0.

Figura 29 – Fractal de Mandelbrot e Julia implementados



Fonte: O Autor (2022).

4.1.2 Implementação sequencial do fractal de Mandelbulb

O Algoritmo 8 mostra o código desenvolvido para gerar o fractal de Mandelbulb. A implementação desse fractal é descrita no trabalho de BINOTTO (2012). A implementação desenvolvida gera um plano do fractal, conforme definido na linha 13. Para o armazenamento do fractal foi utilizado um *array* da biblioteca NumPy (linha 47). Entre as linhas 36 e 42 estão as funções de recorrência que são utilizadas para gerar o fractal de Mandelbulb.

Algoritmo 8 – Mandelbulb em Python

```
1  import numpy as np
2  import math
3  from matplotlib import pyplot as plt
4  from pylab import imshow, show
5  from timeit import default_timer as timer
6
7  def criar_fractal(min_x, max_x, min_y, max_y, imagem, iteracoes):
8      altura = imagem.shape[0]
9      largura = imagem.shape[1]
10     n = 8
11     pi2 = math.pi * 2.0
12     # Angulos de rotacao para converter o plano 2D para o plano 3D
13     xy = 5.0; xz = 5.0; yz = 5.0
14     sxy = math.sin(xy); cxy = math.cos(xy)
15     sxz = math.sin(xz); cxz = math.cos(xz)
16     syz = math.sin(yz); cyz = math.cos(yz)
```

```
17
18      origx = (min_x + max_x) / 2.0
19      origy = (min_y + max_y) / 2.0
20
21      for ky in range(altura):
22          b = ky * (max_y - min_y) / (altura - 1) + min_y
23          for kx in range(largura):
24              a = kx * (max_x - min_x) / (largura - 1) + min_x
25              x = a
26              y = b
27              z = 0.0
28              # Rotacao 3d ao redor do centro do plano
29              x = x - origx; y = y - origy
30              x0 = x * cxy - y * sxy; y = x * sxy + y * cxy
31              x = x0; x0 = x * cxz - z * sxz;z = x * sxz + z * cxz # xy rot
32              x = x0; y0 = y * cyz - z * syz; z = y * syz + z * cyz # xz rot
33              y = y0; x = x + origx; y = y + origy # yz rot
34              cx = x; cy = y; cz = z
35              for i in range(iteracoes):
36                  r = math.sqrt(x * x + y * y + z * z)
37                  t = math.atan2(math.hypot(x, y), z)
38                  p = math.atan2(y, x)
39                  rn = r**n
40                  x = rn * math.sin(t * n) * math.cos(p * n) + cx
41                  y = rn * math.sin(t * n) * math.sin(p * n) + cy
42                  z = rn * math.cos(t * n) + cz
43                  if x * x + y * y + z * z > 4.0:
44                      break
45              imagem[kx, ky] = i
46
47  imagem = np.zeros((4000, 4000), dtype=np.double)
48  inicio = timer()
49  criar_fractal(-1.5, 1.5, -1.5, 1.5, imagem, 1000)
50  tempo_exec = timer() - inicio
51  print(f"Tempo:{tempo_exec}")
52  imshow(imagem)
53  show()
```

Fonte: Adaptado de Binotto (2012)

Na Figura 30 apresenta-se o fractal de Mandelbulb que foi gerado para a face $xy = 5.0$ $xz = 5.0$ $yz = 5.0$. O fractal de Mandelbulb foi gerado utilizando valores de x e de y nos intervalos de -1.5 a 1.5.

Figura 30 – Mandelbulb em 2D

Fonte: O Autor (2022).

4.1.3 Implementação sequencial da árvore browniana

No Algoritmo 9 há o código desenvolvido para gerar o fractal da árvore browniana. Nas linhas 14 e 15 encontram-se as declarações utilizadas para a geração dos pontos aleatórios e entre as linhas 16 e 32 mostra-se o processo em que um ponto é movido aleatoriamente até ser anexado à árvore. As imagens são armazenadas em um *array* da biblioteca NumPy (linha 35).

Algoritmo 9 – Árvore browniana em Python

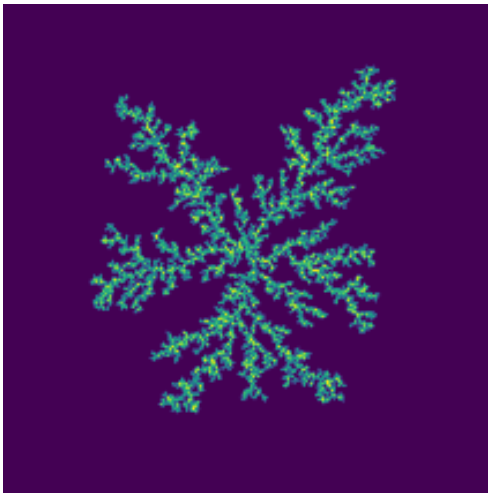
```
1  import numpy as np
2  import random
3  from matplotlib import pyplot as plt
4  from pylab import imshow, show
5  from timeit import default_timer as timer
6
7  def criar_fractal(imagem, iteracoes):
8      altura = int(imagem.shape[0])
9      largura = int(imagem.shape[1])
10     cor = largura / 2
11     imagem[int(altura/2), int(largura/2)] = cor
12
13     for i in range(iteracoes):
14         x = random.randint(1, (largura - 1))
15         y = random.randint(1, (altura - 1))
16         while (1):
17             x_old = x
18             y_old = y
```

```
19                x += random.randint(-1, 1)
20                y += random.randint(-1, 1)
21                if x >= largura:
22                    x = 1
23                if x <= 0:
24                    x = (largura - 1)
25                if y >= altura:
26                    y = 1
27                if y <= 0:
28                    y = (altura - 1)
29                if (imagem[y, x] == cor and y != (altura - 1) and y != 1
30                and x != (largura - 1) and y != 1):
31                    imagem[y_old, x_old] = cor
32                    break
33        return imagem
34
35    imagem = np.zeros((500, 500), dtype=np.uint8)
36    inicio = timer()
37    criar_fractal(imagem, 15000)
38    tempo_exec = timer() - inicio
39    print(f"Tempo:{tempo_exec}")
40    imshow(imagem)
41    show()
```

Fonte: O autor (2022)

A Figura 31 mostra o fractal da árvore browniana criada. A imagem do fractal foi gerada utilizando uma resolução de 500×500 pontos e foram anexados à árvore um total de 15.000 pontos.

Figura 31 – Árvore browniana



Fonte: O Autor (2022).

5 PARALELIZAÇÃO DOS FRACTAIS

Existem diferentes alternativas para a utilização da biblioteca CUDA na linguagem de Programação Python, sendo que entre elas destacam-se o *wrapper*[1] CUDA Python (NVIDIA, 2022) e a biblioteca Numba (NUMBA, 2022a).

5.1 CUDA PYTHON

Uma primeira alternativa para utilização de CUDA na linguagem de programação Python é o CUDA Python. Ele consiste em um *wrapper* que foi criado pela própria empresa NVIDIA. No Algoritmo 10 mostra-se um exemplo de um programa desenvolvido em CUDA Python. A *string saxpy* possui o código CUDA, em linguagem C (linha 2). O código existente na *string saxpy* será compilado para, posteriormente, ser executado na GPU. Em CUDA Python, todo o código dirigido ao *device* deve ser desenvolvido nas linguagens de programação C/C++.

Algoritmo 10 – CUDA em Python

```
1  # Cria string em C para complilacao
2  saxpy = '''\
3  extern "C" __global__
4  void saxpy(float a, float *x, float *y, float *out, size_t n)
5  {
6      size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
7      if (tid < n) {
8          out[tid] = a * x[tid] + y[tid];
9      }
10 }
11 '''
12 # Cria o programa
13 prog = nvrtc.nvrtcCreateProgram(str.encode(saxpy), b'saxpy.cu',
14 0, [], [])
15 # Compila o programa
16 prog_comp = nvrtc.nvrtcCompileProgram(prog, len(opts), opts)
17 # Pega os dados da compilacao
18 data_prog = nvrtc.nvrtcGetPTX(prog, data)
19 # Carrega os dados como modulo e recupera a funcao
20 data = np.char.array(data)
21 module = cuda.cuModuleLoadData(data)
22 kernel = cuda.cuModuleGetFunction(module, b'saxpy')
```

Fonte: Adaptado de Nvidia (2022)

[1] Consiste em um padrão de projeto para adaptar uma *interface* de uma classe para uma outra *interface*.

A compilação do código existente na *string saxpy* é realizada em tempo de execução através da biblioteca *NVIDIA Runtime Compilation Library* (NVRTC). A função *nvrtcCreateProgram* (linha 13) gera um arquivo CUDA (extensão *.cu*) com o código existente na *string* passada como parâmetro. A função *nvrtcCompileProgram* compila o arquivo CUDA utilizando o NVRTC (linha 27). Por fim, a função *nvrtcGetPTX* recebe o arquivo compilado e armazena na váriavel *ptx*. Posteriormente, o código compilado é utilizado como um módulo, carregado em tempo de execução (linha 22).

Como pode ser observado no Algoritmo 10, para a utilização do *wrapper* do CUDA Python, todo o código para a programação CUDA deve ser desenvolvido através de *strings* inseridas no código Python. Essas *strings* devem apresentar o código das funções *kernels* implementadas nas linguagens de programação C ou C++. Dessa forma, o programador deve ter conhecimento sobre a utilização de CUDA nessas linguagens.

5.2 BIBLIOTECA NUMBA

A biblioteca Numba é um compilador JIT (do inglês *Just in time*)[2] que converte um subconjunto da linguagem Python e da biblioteca NumPy para código executável. O Numba utiliza o CPython, que é interpretador padrão do Python, como interpretador e o compilador LLVM (*Low Level Virtual Machine*) para gerar o código executável. O LLVM é um compilador e otimizador que se encontra presente na maioria das máquinas virtuais que possuem um JIT.

O funcionamento do Numba baseia-se em um processo de compilação das funções desenvolvidas em Python. Nesse caso, é realizada a leitura do código da função em *bytecodes*, gerando um código compilado através do LLVM. A versão compilada é utilizada sempre que a função for chamada no programa (NUMBA, 2022b).

O código da função em *bytecodes* pode ser combinado ainda com informações sobre os tipos de dados utilizados. As funções do Python são projetadas para operarem sobre tipos de dados genéricos, o que as tornam mais flexíveis, porém mais lentas. Durante o processo de compilação, o Numba realiza uma etapa de inferência de tipos, através do qual determina os tipos das variáveis utilizadas na função (NUMBA, 2022b). Além disso, é possível o programador informar o tipo das variáveis na declaração do decorador.

[2] Compilação de um programa em tempo de execução

O Numba é flexível, permitindo a geração de código para diferentes arquiteturas, como CPUs e GPUs. Ele oferece opções para a geração de código Python para CPUs e GPUs a partir de uma coleção de decoradores[3]. O Algoritmo 11 apresenta o código para o cálculo da hipotenusa utilizando a biblioteca Numba. A linha 4 possui o decorador que gera código nativo para ser executado em uma CPU. Como pode ser observado, foram informados os tipos das variáveis utilizados na função, como inteiros de 32 *bits* (tipo $i4$). Além disso, foi especificado, através do uso do tipo $f8$, que o valor de retorno é um *double*.

Algoritmo 11 – Função Numba para calcular a hipotenusa

```
1  from numba import jit
2  import math
3
4  @jit('f8(i4,i4)')
5  def hypot(x, y):
6      x = abs(x);
7      y = abs(y);
8      t = min(x, y);
9      x = max(x, y);
10     t = t / x;
11     return x * math.sqrt(1+t*t)
```

Fonte: (SEIBERT; LAM, 2017)

O Numba possui suporte para a programação CUDA em GPU, compilando um subconjunto de código Python em funções *kernels*. Nesse caso, o Numba converte o código Python para o *framework* CUDA na linguagem C. As funções *Kernels* possuem acesso direto aos *arrays* da biblioteca NumPy, visto que esses são transferidos automaticamente para a GPU. No Algoritmo 12 há um exemplo de uma função *kernel* que multiplica por 2 um vetor de entrada. A declaração do *kernel* é realizada através do decorador *@cuda.jit* (linha 7). Na chamada da função *kernel* (linha 17) é informado o número de *threads* por bloco (definido na linha 15) e o número de blocos por *grid* (definido na linha 16). A função *cuda.grid* (linha 9) retorna a posição da *thread* na *grid* de blocos.

[3] Consiste em uma anotação que é utilizada na definição de uma função, método ou classe que substitui o elemento por um outro

Algoritmo 12 – Numba CUDA exemplo

```
1  from __future__ import division
2  from numba import cuda
3  import numpy
4  import math
5
6  # CUDA kernel (device)
7  @cuda.jit
8  def my_kernel(io_array):
9      pos = cuda.grid(1)
10     if pos < io_array.size:
11         io_array[pos] *= 2
12 # Codigo Host
13 data = numpy.ones(256)
14 threadsporbloco = 256
15 blocosporgrid = math.ceil(data.shape[0] / threadsporbloco)
16 my_kernel[blocosporgrid, threadsporbloco](data)
17 print(data)
```

Fonte: O autor (2022)

O Numba apresenta recursos que possibilitam a transferência de dados entre o *host* e o *device* (Algoritmo 13). Na *linha* 1 é utilizado o método *device_array* que aloca um *array* na memória do *device*. A *linha* 2 apresenta a função *to_device* que transfere o conteúdo de um *array* para a memória do *device*. A *linha* 3 apresenta a função *copy_to_host* que é usada para transferir um *array* da memória do *device* para o *host*. A *linha* 4 mostra a função *shared.array*, utilizada para a alocação de um *array* compartilhado na memória do *device*. Por fim, a *linha* 5 apresenta a função *syncthreads* que realiza a sincronização (barreira) das *threads*.

Algoritmo 13 – Manuseio de memória em Numba CUDA

```
1 device_array = cuda.device_array(shape)
2 device_array = cuda.to_device(array)
3 host_array = device.copy_to_host()
4 shared_array = cuda.shared.array(shape,type)
5 cuda.syncthreads()
```

Fonte: O autor (2022)

Para o desenvolvimento deste trabalho optou-se pela utilização da biblioteca Numba, visto que ela disponibiliza recursos que permitem a paralelização em GPU, utilizando somente a linguagem de programação Python. Na biblioteca CUDA Python, o código para a GPU deve ser desenvolvido utilizando *strings* com o código nas linguagens de programação C ou C++. Dessa forma, o programador deve ter conhecimento sobre programação CUDA nessas linguagens.

5.3 PARALELIZAÇÃO DOS FRACTAIS DE MANDELBROT E JULIA

A paralelização dos fractais foi realizada dividindo os laços responsáveis pela geração de cada ponto da imagem (*pixel*). O *grid* foi declarado em duas dimensões, visto que a imagem se trata de um *array* de duas dimensões. O *grid* foi dividido em blocos de *threads*, onde o tamanho de cada bloco foi obtido a partir da razão entre largura e altura da imagem pelo número de *threads* (linha 3 e 4 do Algoritmo 14). Em todas as paralelizações utilizou-se um número de *threads* igual a 256 (16×16) (linha 2), organizadas em blocos de duas dimensões por ser uma imagem bidimensional. A escolha por 256 *threads* foi devido ao fato de este valor ser divisível por 32, considerando que os *kernels* atribuem as instruções a *warps* de 32 *threads*.

Algoritmo 14 – Declaração das variáveis utilizadas no CUDA

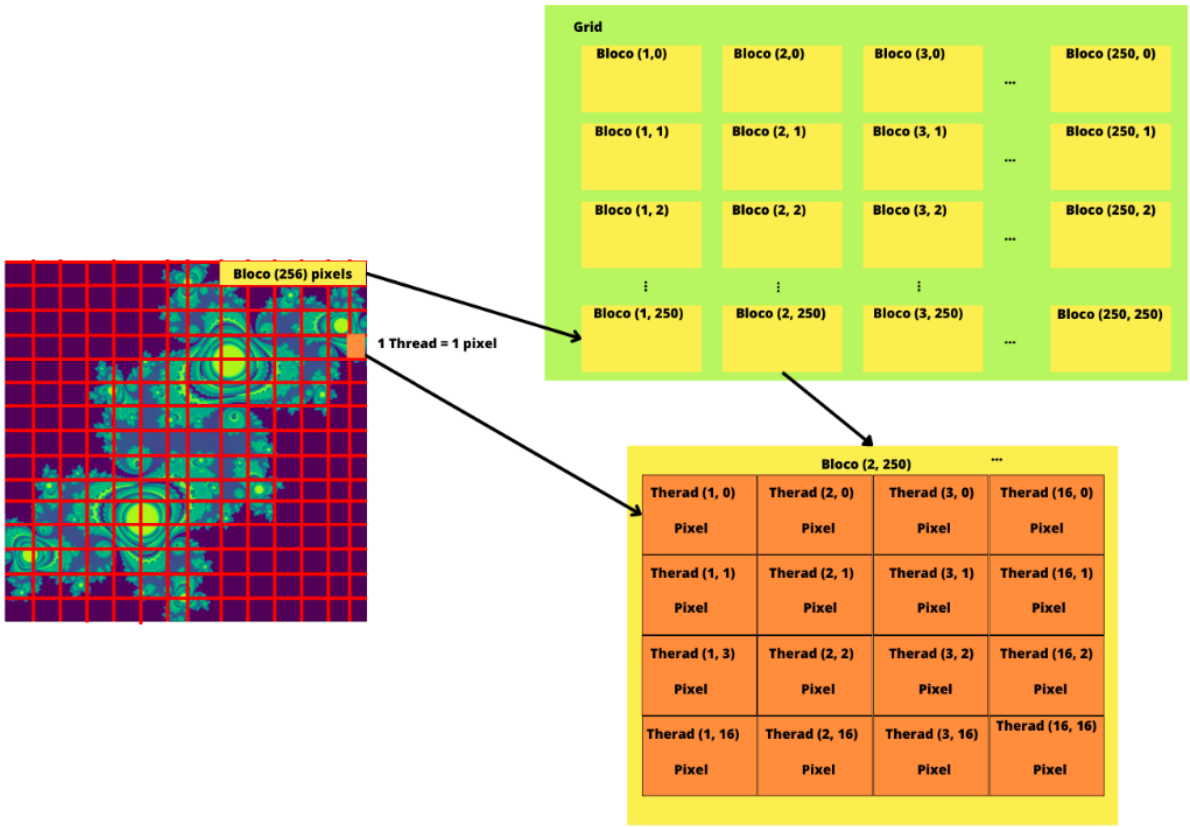
```
1 imagem = np.zeros((4000, 4000), dtype = np.uint8)
2 threads_por_bloco = (16, 16)
3 blocos_por_grid_x = math.ceil(imagem.shape[0] / threads_por_bloco[0])
4 blocos_por_grid_y = math.ceil(imagem.shape[1] / threads_por_bloco[1])
5 blocos_por_grid = (blocos_por_grid_x, blocos_por_grid_y)
6 imagem_a = cuda.to_device(imagem)
7 mandelbrot_kernel[blocos_por_grid, threads_por_bloco](-2.0, 1.0, -1.0, 1.0,
8 imagem_a, 20)
9 imagem = imagem_a.copy_to_host()
```

Fonte: O autor (2022)

No Algoritmo 14, podem ser observadas ainda as funções utilizadas para a manipulação de memória. Para a execução do código, é necessário alocar um *array* para o armazenamento da imagem na memória da GPU (*device*) (linha 6). Da mesma forma, ao final é necessário copiar a imagem resultante da memória da GPU para a memória principal (*host*) para visualização das imagens (linha 9).

A Figura 32 mostra uma representação da estrutura utilizada para a paralelização dos fractais. Neste exemplo foi considerada a geração de um fractal com 4.000×4.0000 pontos. Como pode ser observado, a geração do fractal foi dividida em um *grid*, com 250×250 blocos de *threads*. O número de blocos *threads* foi obtido a partir da divisão da largura e da altura da imagem pelo número de *threads* de cada bloco (256 *threads*). Assim, cada bloco de *thread* será responsável pelo cálculo de 256 pontos do fractal.

Figura 32 – Estrutura da paralelização



Fonte: O Autor (2022).

A chamada da função *kernel* pode ser observada na linha 7 do Algoritmo 14. A função *kernel* tem como entrada o número de *threads* por bloco e o número de blocos (linha 2 e linha 5). O Algoritmo 15 apresenta o código da função *kernel* para o fractal de Mandelbrot. Como pode ser observado na linha 1, o decorador especifica os tipos dos parâmetros de entrada da função (linha 2). As variáveis *gridX* e *gridY* são utilizadas para incrementar os laços responsáveis por gerar os pontos. Assim, cada bloco de *threads* é responsável por gerar 256 pontos do fractal. Quando a execução de bloco de *threads* termina, os próximos 256 pontos são atribuídos a um próximo de bloco de *threads*, e assim sucessivamente até que se gerem todos os pontos do fractal.

Algoritmo 15 – Implementação do *Kernel* de Mandelbrot

```
1  @cuda.jit((f8, f8, f8, f8, uint8[:,:], uint32))
2  def mandelbrot_kernel(min_x, max_x, min_y, max_y, imagem, iteracoes):
3      altura = imagem.shape[0]
4      largura = imagem.shape[1]
5
6      tamanho_pixel_x = (max_x - min_x) / largura
7      tamanho_pixel_y = (max_y - min_y) / altura
8
9      startX, startY = cuda.grid(2)
10     gridX = cuda.gridDim.x * cuda.blockDim.x
11     gridY = cuda.gridDim.y * cuda.blockDim.y
12
13     for x in range(startX, largura, gridX):
14         real = min_x + x * tamanho_pixel_x
15         for y in range(startY, altura, gridY):
16             imag = min_y + y * tamanho_pixel_y
17             imagem[y, x] = mandelbrot_gpu(real, imag, iteracoes)
```

Fonte: O autor (2022)

As implementações de Mandelbrot e Julia utilizam a mesma função *kernel*, alterando somente a função que calcula a equação de recorrência de cada fractal. Algoritmo 16 apresenta a função que realiza o cálculo da equação de recorrência do fractal de Mandelbrot. Na linha 11 é realizada a chamada para a execução da função na GPU.

Algoritmo 16 – Implementação de Mandelbrot e chamada para GPU

```
1  @jit
2  def mandelbrot(x, y, iteracoes):
3      c = complex(x, y)
4      z = 0.0j
5      for i in range(iteracoes):
6          z = z * z + c
7          if (z.real * z.real + z.imag * z.imag) >= 4:
8              return i
9      return iteracoes
10
11 mandelbrot_gpu = cuda.jit(uint32(f8, f8, uint32), device=True)(mandelbrot)
```

Fonte: O autor (2022)

5.4 PARALELIZAÇÃO DO FRACTAL DE MANDELBULB

Para a paralelização do fractal de Mandelbulb utilizou-se o mesmo procedimento descrito na paralelização do fractal de Mandelbrot e de Julia. Optou-se por gerar somente um plano deste fractal, pois a biblioteca Numba CUDA não possui suporte para diversos métodos de *arrays* tridimensionais da biblioteca NumPy (NUMBA, 2022b). A escolha do plano é realizada na linha 7 do Algoritmo 17.

Algoritmo 17 – Implementação de Mandelbulb para GPU

```
1  @cuda.jit
2  def criar_fractal(min_x, max_x, min_y, max_y, imagem, iteracoes):
3      altura = imagem.shape[0]
4      largura = imagem.shape[1]
5      n = 8
6      pi2 = math.pi * 2.0
7      xy = 5.0; xz = 5.0; yz = 5.0
8      sxy = math.sin(xy); cxy = math.cos(xy)
9      sxz = math.sin(xz); cxz = math.cos(xz)
10     syz = math.sin(yz); cyz = math.cos(yz)
11     origx = (min_x + max_x) / 2.0; origy = (min_y + max_y) / 2.0
12     startX, startY = cuda.grid(2)
13     gridX = cuda.gridDim.x * cuda.blockDim.x
14     gridY = cuda.gridDim.y * cuda.blockDim.y
15     for ky in range(startY, width, gridY):
16         b = ky * (max_y - min_y) / (altura - 1) + min_y
17         for kx in range(startX, width, gridX):
18             a = kx * (max_x - min_x) / (largura - 1) + min_x
19             x = a; y = b; z = 0.0
20             x = x - origx; y = y - origy
21             x0 = x * cxy - y * sxy; y = x * sxy + y * cxy
22             x = x0; x0 = x * cxz - z * sxz;z = x * sxz + z * cxz # xy rot
23             x = x0; y0 = y * cyz - z * syz; z = y * syz + z * cyz # xz rot
24             y = y0; x = x + origx; y = y + origy # yz rot
25             cx = x; cy = y; cz = z
26             for i in range(iteracoes):
27                 r = math.sqrt(x * x + y * y + z * z)
28                 t = math.atan2(math.hypot(x, y), z)
29                 p = math.atan2(y, x)
30                 rn = r**n
31                 x = rn * math.sin(t * n) * math.cos(p * n) + cx
32                 y = rn * math.sin(t * n) * math.sin(p * n) + cy
33                 z = rn * math.cos(t * n) + cz
34                 if x * x + y * y + z * z > 4.0:
35                     break
36             imagem[kx, ky] = i
```

Fonte: O autor (2022)

5.5 PARALELIZAÇÃO DO FRACTAL DA ÁRVORE BROWNIANA

A paralelização do fractal árvore browniana foi realizada através da divisão do processo de geração dos pontos e, aqui, foi utilizado um *grid* de uma única dimensão. Diferentemente dos demais fractais, onde toda a imagem é percorrida gerando um valor para cada ponto da imagem, nesse caso é gerado um ponto que é movimentado até se anexar à árvore. Assim, quando a execução de um bloco de *threads* tem início, cada *thread* do bloco gera um ponto a ser anexado à árvore. Quando a execução do bloco termina, outro bloco de *threads* é executado, gerando outros 256 pontos a serem anexados. Esse processo é repetido até ser atingido um número máximo de pontos. Por se tratar de um fractal com geração aleatória de pontos, foi utilizada a função *xoroshiro128*, que gera números aleatórios utilizando os *cores* da GPU. No Algoritmo 18 apresenta-se o código paralelizado do fractal da árvore browniana.

Algoritmo 18 – Implementação da árvore browniana na GPU

```
@cuda.jit
def criar_fractal(imagem, iteracoes, rng_states):
    altura = int(imagem.shape[0])
    largura = int(imagem.shape[1])
    cor = largura / 2
    imagem[int(altura/2), int(largura/2)] = cor

    startX = cuda.grid(1)
    gridX = cuda.gridDim.x * cuda.blockDim.x
    cuda.syncthreads()

    for i in range(startX, iteracoes, gridX):
        x = xoroshiro128p_uniform_float32(rng_states, startX)
        x = int(math.floor(x) * largura)
        y = xoroshiro128p_uniform_float32(rng_states, startX)
        y = int(math.floor(y) * largura)
        while(1):
            x_old = x
            y_old = y
            alea2 = int(xoroshiro128p_uniform_float32(rng_states,
            startX) * 4)
            if(alea2 == 0):
                x = x + int(xoroshiro128p_uniform_float32(rng_states,
                startX) * 2)
                y = y + int(xoroshiro128p_uniform_float32(rng_states,
                startX) * 2)
            if(alea2 == 1):
                x = x + int(xoroshiro128p_uniform_float32(
                        rng_states,
                        startX) * 2) * -1
                y = y + int(xoroshiro128p_uniform_float32(rng_states,
```

```
32                      startX) * 2)
33              if(alea2 == 2):
34                  x = x + int(xoroshiro128p_uniform_float32(rng_states,
35                      startX) * 2)
36                  y = y + int(xoroshiro128p_uniform_float32(
37                          rng_states,
38                          startX) * 2) * -1
39              if(alea2 == 3):
40                  x = x + int(xoroshiro128p_uniform_float32(
41                          rng_states,
42                          startX) * 2) * - 1
43                  y = y + int(xoroshiro128p_uniform_float32(
44                          rng_states,
45                          startX) * 2) * - 1
46              if x >= largura:
47                  x = 1
48              if x <= 0:
49                  x = (largura - 1)
50              if y >= altura:
51                  y = 1
52              if y <= 0:
53                  y = (altura - 1)
54              if (imagem[y, x] == cor and y != (altura - 1) and y != 1
55              and x != (largura - 1) and y != 1):
56                  imagem[y_old, x_old] = cor
57                  break
58
59
60  imagem = np.zeros((500, 500), dtype=np.uint8)
61  threads_por_bloco = 256
62  blocos_por_grid = 65500
63  rng = create_xoroshiro128p_states(threads_por_bloco * blocos_por_grid,
64                                    seed=1)
65  inicio = timer()
66  imagem_a = cuda.to_device(imagem)
67  criar_fractal[blocos_por_grid, threads_por_bloco](imagem_a, 15000, rng)
68  imagem = imagem_a.copy_to_host()
69  tempo_exec = timer() - inicio
70  print(f"Tempo:{tempo_exec}")
71  imshow(imagem)
72  show()
```

Fonte: O autor (2022)

6 RESULTADOS OBTIDOS

Todos os testes foram realizados utilizando um computador com um processador *AMD Ryzen 5 5600X* com 6 núcleos de processamento, com uma frequência de 3.7 GHz. O processador também conta com uma *cache L1* de 384 KB, cache L2 de 3 MB e *cache L3* de 32 MB. Além disso, o computador possui memória RAM DDR4 de 16 GB, operando a 3200 MHz, e dois discos rígidos SSD de 480 GB cada. O sistema operacional é o *Ubuntu 22.04 LTS* de 64 *bits*. A placa gráfica utilizada para os testes foi uma *GeForce® GTX 1660 SUPER™ OC 6G* da NVIDIA, que pode ser observada na Figura 33. Essa possui 1.408 CUDA *cores* distribuídos em 22 *Streaming Multiprocessor*. A memória global é 6 GB, com uma interface de 192 *bits* e a largura de banda é de 336 GB por segundo. Essa GPU possui ainda 56 unidades de textura.

Figura 33 – GeForce® GTX 1660 SUPER™ OC 6G

Fonte: Gigabyte (2019).

6.1 IMPLEMENTAÇÕES EM PYTHON

Ao todo, foram implementados quatro fractais: Mandelbrot, Julia, Mandelbulb e a árvore browniana. Todos os fractais foram implementados na linguagem de programação Python versão 3.0. Para cada fractal foram desenvolvidas 3 versões: a primeira delas utilizando somente a linguagem Python; a segunda utilizando a biblioteca Numba, que realizava uma conversão dos *bytecodes* para linguagem de máquina antes da execução; e, por fim, uma versão que utiliza a biblioteca Numba em conjunto com o *framework* CUDA.

As implementações, com exceção da árvore browniana, foram testadas utilizando uma imagem com a resolução de 4.000×4.000. O número máximo de iterações foi de 1.000 para todos os fractais. Para gerar a árvore browniana foi utilizada uma imagem com uma resolução de 500×500 e foram gerados 15.000 pontos. No caso da árvore browniana, a utilização de uma resolução de 4.000×4.000 resultaria em uma árvore com dimensões muito pequenas.

A Tabela 1 apresenta os tempos de execução obtidos (em segundos) para cada um dos fractais. O resultado apresentado é a média aritmética de 10 execuções. Como pode ser observado, as implementações utilizando Numba sem suporte à GPU apresentaram um tempo de execução inferior em relação às implementações originais. De fato, a implementação de Mandelbrot em Numba foi 51 vezes mais rápida que a original em Python. As implementações em Numba dos fractais de Julia e de Mandelbulb foram, respectivamente, 50 vezes e 12 vezes mais rápidas que as versões originais. Por fim, a implementação da árvore browniana foi 241 vezes mais rápida que a versão original.

Tabela 1 – Resultados obtidos em Python em segundos

Fractal	**Python**	**Numba**	**CUDA**
Mandelbrot	498,81	9,86	0,67
Julia	297,45	5,99	0,58
Mandelbulb	2837,41	241,77	11,92
Árvore browniana	1932,82	8,11	0,73

Fonte: O Autor (2022).

As implementações originais apresentaram um baixo desempenho devido ao fato de Python ser uma linguagem não compilada. Em Python, inicialmente o código é convertido para *bytecodes*, que posteriormente são interpretados. Além disso, o Python é uma linguagem de tipagem dinâmica, que é um processo que apresenta um elevado custo computacional. Já em Numba, as funções que apresentam o decorador são compiladas resultando em uma diminuição significativa no tempo de execução. Ademais, o uso de decoradores possibilita ao desenvolvedor informar os tipos dos dados, evitando, assim, a necessidade da tipagem dinâmica.

As implementações em CUDA Numba também apresentaram tempos inferiores aos obtidos utilizando somente a biblioteca Numba. A implementação do fractal de Mandelbrot utilizando CUDA Numba foi 15 vezes mais rápida que a versão que usa Numba para CPU. Já as implementações dos fractais de Júlia, Mandelbulb e árvore browniana para GPU foram, respectivamente, 10 vezes, 20 vezes e 11 vezes mais rápidas que as versões que usam Numba para CPU. Dessa forma, conclui-se que a utilização de Numba com suporte à GPU é uma forma eficiente de explorar o paralelismo em GPUs na linguagem Python.

6.2 COMPARAÇÃO COM IMPLEMENTAÇÕES EM LINGUAGEM C

Testes foram realizados para verificar o desempenho das implementações em Numba, quando comparadas a linguagens compiladas. Para isso, os resultados obtidos com Numba foram comparados com implementações desenvolvidas utilizando a linguagem de programação C e paralelizadas com a biblioteca de *threads* OpenMP. O OpenMP é um padrão utilizado para a paralelização de aplicações em ambientes de memória compartilhada.

A Tabela 2 apresenta uma comparação entre tempos de execução (em segundos) das implementações em Numba com as implementações desenvolvidas em linguagem C. As versões em OpenMP foram executadas com 6 *threads*, visto que esse era o número de núcleos físicos disponíveis no computador utilizado para os testes. Todas as versões foram compiladas utilizando o compilador *gcc*, com a opção de otimização -03.

Tabela 2 – Resultados obtidos em Linguagem C em segundos

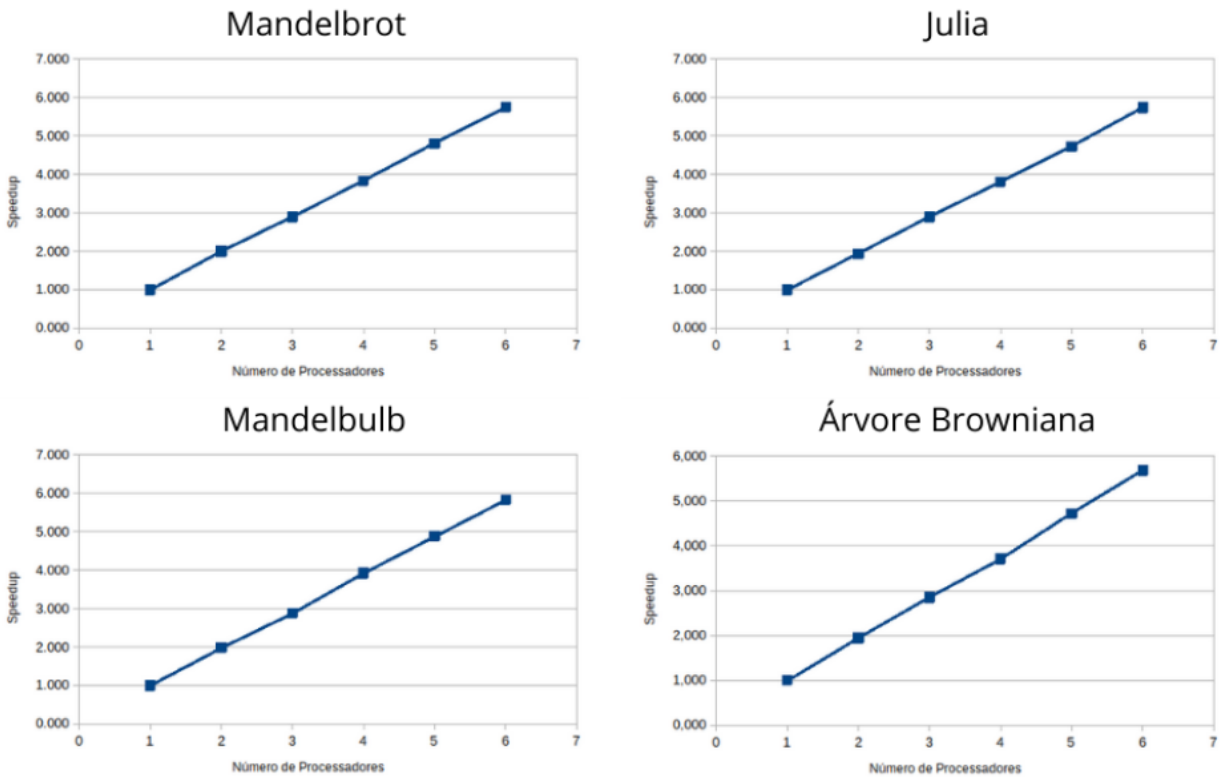
Fractal	**Numba CPU**	**C - Sequencial**	**C - 6 *threads***
Mandelbrot	9,86	9,75	1,68
Julia	5,99	5,86	1,01
Mandelbulb	241,77	237,54	42,13
Árvore browniana	8,11	7,49	1,51

Fonte: O Autor (2022).

Nos gráficos da Figura 34 são apresentados os *speedups*[1] obtidos para as versões paralelizadas utilizando a biblioteca OpenMP. Como pode ser observado, as implementações paralelizadas apresentaram um bom desempenho. De fato, observa-se nos gráficos que implementações possuem um *speedup* praticamente linear. A partir disso, conclui-se que os recursos computacionais estão sendo utilizados eficientemente pelas implementações paralelas.

[1] Indica quantas vezes o programa paralelo é mais rápido que a versão sequencial, sendo calculado pela razão entre o melhor tempo sequencial e o melhor tempo da versão paralela

Figura 34 – *Speedup* dos programas paralelizados em OpenMP



Fonte: O Autor (2022).

7 CONSIDERAÇÕES FINAIS

Neste trabalho foi desenvolvido um estudo sobre a paralelização de fractais, utilizando CUDA na linguagem de programação Python. A linguagem de programação Python foi escolhida devido ao crescimento na sua utilização nos últimos anos. Esse crescimento é consequência, principalmente, da facilidade de aprendizado e do grande conjunto de bibliotecas e pacotes disponíveis na internet, para as mais diversas aplicações.

A arquitetura de GPU foi escolhida devido ao crescimento na utilização desse tipo de arquitetura para a execução de aplicações de uso geral. Além disso, essa arquitetura já é comum em computadores pessoais em decorrência do crescente uso dos computadores pessoais para jogos e outras aplicações gráficas 3D. Optou-se pela utilização do *framework* CUDA, uma vez que esse foi desenvolvido especificamente para o desenvolvimento de aplicações para as GPUs da empresa NVIDIA, que é a fabricante da GPU que foi utilizada para o desenvolvimento e para a realização dos testes das implementações que serão realizadas.

A linguagem Python não é amplamente utilizada para a exploração de paralelismo em GPUs, apresentando poucos estudos que abordem especificamente esse assunto. Dessa maneira, esse trabalho consistiu em um estudo inicial sobre a utilização do *framework* CUDA na linguagem de programação Python, podendo ser utilizado como uma referência básica para o desenvolvimento de outras aplicações.

Optou-se pela paralelização dos fractais de Julia, Mandelbrot e Mandelbulb, pois estes são muito utilizados em avaliações de ambientes e ferramentas de programação paralela. Eles são gerados a partir de funções de recorrência, sendo que cada ponto no espaço pode ser calculado de forma independente, sendo assim de fácil paralelização. O fractal da árvore browniana foi escolhido por ser gerado a partir de posições aleatórias, apresentando diferenças no processo de paralelização.

A biblioteca Numba foi escolhida pois permite que todo o desenvolvimento seja realizado utilizando somente a linguagem de programação Python. Nesse caso, a paralelização é realizada através do uso de um conjunto de decoradores, sem a necessidade de se conhecer uma outra linguagem de programação. Já a biblioteca CUDA Python, que é disponibilizada pela NVIDIA, consiste apenas em um *wrapper*. Assim, o programador necessita conhecer as linguagens de programação C ou C++, para o desenvolvimento do código para CUDA.

Os resultados obtidos mostram que a utilização da biblioteca Numba possibilita o desenvolvimento de programas em linguagem Python com desempenhos próximos aos obtidos com uma linguagem compilada. De fato, observa-se que os tempos obtidos utilizando Numba foram similares aos obtidos utilizando a linguagem de programação C. Da mesma forma, as implementações desenvolvidas utilizando Numba CUDA mostraram que é possível explorar de forma eficiente o paralelismo de GPUs em Python. Entretanto, nem todas as funcionalidades estão disponíveis em Numba, como o caso de funções de álgebra linear para o processamento tridimensional, o que pode dificultar o desenvolvimento de algumas aplicações.

7.1 TRABALHOS FUTUROS

Como sugestão de possíveis trabalhos futuros propõe-se:

- Paralelização do fractal Mandelbox, um fractal 3D que representa as propriedades do fractal de Mandelbrot sem ser em duas dimensões;
- Um estudo sobre a utilização do *wrapper* CUDA Python para a paralelização de cálculos fractais;
- Um estudo comparativo entre Numba CUDA e a utilização de CUDA nas linguagens de programação C e C++.

REFERÊNCIAS

ALFIO. **Brownian Tree**. Rosetta Code, 2003. Algoritmos da árvore browniana em diversas linguagens de programação. Disponível em: <https://rosettacode.org/wiki/Brownian_tree>. Acesso em: 5 jun. 2022.

ARSIE, K. C. **Dimensão Espacial**. Paraná: Universidade Federal do Paraná, 2007. Discute o conceito de dimensão espacial. Disponível em: <https://docs.ufpr.br/~ewkaras/ic/karla08.pdf>. Acesso em: 13 jun. 2022.

ASSIS, T. A. d. *et al.* Geometria fractal: propriedades e características de fractais ideais. **Revista Brasileira de Ensino de Física**, scielo, v. 30, p. 2304.1–2304.10, 2008. Disponível em: <http://www.scielo.br/scielo.php?script=sci\%5Farttext&pid=S1806-11172008000200005&lang=pt>.

BAKER, B. **The Rise of Parallel Computing**. SORT Conference, 2011. Slides apresentados na SORT Conference de 2011. Disponível em: <https://www.slideshare.net/bakers84/the-rise-of-parallel-computing>. Acesso em: 15 jun. 2022.

BASU, S. K. **Parallel and Distributed Computing**. Delhi, India: PHI Learning, 2016.

BINOTTO, F. **Mandelbulb Fractal (Python Recipe)**. ActiveState Code, 2012. Disponível em: <https://code.activestate.com/recipes/578198-mandelbulb-fractal/>.

BOURKE, P. **DLA - Diffusion Limited Aggregation**. Paul Bourke, 2014. Explicação sobre DLA - Diffusion Limited Aggregation (árvore browniana). Disponível em: <http://paulbourke.net/fractals/dla/>. Acesso em: 21 mai. 2022.

BRADBURY, R.; REICHELT, R.; GREEN, D. Fractals in Ecology: Methods and Interpretation. **Marine Ecology-progress Series - MAR ECOL-PROGR SER**, v. 14, p. 295–296, 01 1984.

CAMPOS, A. M. de. **Animated Fractal Mountain**. Campos, 2006. GIF da montanha fractal. Disponível em: <https://commons.wikimedia.org/wiki/File:Animated_fractal_mountain.gif>. Acesso em: 5 jun. 2022.

CHAPMAN, B.; JOST, G.; PAS, R. van der. **Using OpenMP**. London, England: MIT Press, 2007. (Scientific and Engineering Computation).

CORSO, T. B. **Crux: Ambiente Multicomputador Configurável por Demanda**. Universidade Federal de Santa Catarina, 1999. Disponível em: <https://repositorio.ufsc.br/bitstream/handle/123456789/81058/147365.pdf?sequence=1>. Acesso em: 2 jun. 2022.

CUDA Toolkit Documentation. Nvidia, 2022. Documentação CUDA em C++. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Acesso em: 2 jun. 2022.

CUNHA, R. D. da. Computação paralela: Uma Breve Introdução. In: XX CONGRESSO NACIONAL DE MATEMATICA APLICADA E COMPUTACIONAL CNMAC, 1997, Gramado. Porto Alegre: Sbmac, 1997. Disponível em: <https://www.lume.ufrgs.br/bitstream/handle/10183/181429/000459402.pdf?sequence=1>. Acesso em: 22 mai. 2022.

DRAKOPOULOS, V.; MIMIKOU, N.; THEOHARIS, T. An Overview of Parallel Visualisation Methods for Mandelbrot and Julia sets. **Computers and Graphics**, v. 27, n. 4, p. 635–646, 2003. ISSN 0097-8493. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0097849303001067>.

FERGUSON, K. **Fractal Antennas**. Stanford: Stanford University, 2012. Submissão para o curso Physics of the Cell Phone. Disponível em: <http://large.stanford.edu/courses/2012/ph250/ferguson1/>. Acesso em: 5 jun. 2022.

FERNANDES, T. **Legado sem Fronteiras**. Rio de Janeiro: Ciência Hoje, 2010. Texto sobre Mandelbrot e fractais. Disponível em: <https://cienciahoje.org.br/legado-sem-fronteiras/>. Acesso em: 5 jun. 2022.

FEY, F.; ROSA, J. A. da. Teoria do Caos: A Ordem da não-linearidade. **Universo Acadêmico**, Taquara, v. 5, n. 1, p. 217–232, 2012.

FILHO, S. L. M. **A Curva de Koch (Fractal Floco de Neve)**. Paraná: Paraná Inteligência Artificial, 2002. Apresenta o fractal Floco de Neve. Disponível em: <http://www.batebyte.pr.gov.br/Pagina/Curva-de-Koch-Fractal-Floco-de-Neve>. Acesso em: 5 jun. 2022.

FUZZO, R. A. Fractais: Algumas Características e Propriedades. IV EPCT - Encontro de Produção Científica e Tecnológica, Campo Mourão, 2009. ISSN 1981-6480.

INTEL, 2021, San Francisco. **Intel Innovation**. San Francisco: Intel, 2021.

GIGABYTE. **GeForce® GTX 1660 Super™ OC 6G características: Placas de Vídeo - GIGABYTE Brazil**. GIGABYTE, 2019. Especificação da GTX 1660 Super. Disponível em: <https://www.gigabyte.com/br/Graphics-Card/GV-N166SOC-6GD#kf>. Acesso em: 5 jun. 2022.

GÓMEZ, E. S. MPI vs OpenMP: A Case Study on Parallel Generation of Mandelbrot Set. **Innovación y Software**, v. 1, n. 2, p. 12–26, 2020.

GROPP, W. *et al.* **Using MPI**. London, England: MIT Press, 1995. (Scientific and Engineering Computation).

GäNG, I. K. *et al.* **Parallel Implementation and Analysisof Mandelbrot Set Construction**. Hattiesburg: University of Southern Mississippi, 2008.

CONSUMER TECHNOLOGY ASSOCIATION, 2019, Las Vegas. **Consumer Electronic Show**.

HUNGILO, G. G.; EMMANUEL, G.; PRANOWO. Performance Comparison in Simulation of Mandelbrot Set Fractals Using Numba. In: **The 5th International Conference On Industrial, Mechanical, Electrical, And Chemical Engineering 2019 (Icimece 2019)**. AIP Publishing, 2020. Disponível em: <https://doi.org/10.1063/5.0000636>.

HUSEINOVIć, A.; RIBIć, S. Benchmark Comparison of Computing the Mandelbrot Set in OpenCL. In: **2015 23rd Telecommunications Forum Telfor (TELFOR)**. [S.l.: s.n.], 2015. p. 994–997.

JOURY, A. **Why Python is not the Programming Language of the Future**. Towards Data Science, 2020. Disponível em: <https://towardsdatascience.com/why-python-is-not-the-programming-language-of-the-future-30ddc5339b66>. Acesso em: 13 jun. 2022.

KINDRATENKO, V. (Ed.). **Numerical Computations with GPUs**. Cham, Switzerland: Springer International Publishing, 2014.

KREUTZ, D. L.; KEPLER, F. N.; STEIN, B. **Otimizando o Desempenho de Aplicações de Cálculo de Fractais em Máquinas Multiprocessadas e Aglomerados de Computadores**. Santa Maria: Universidade Federal de Santa Maria, 2019.

LEMES, A. F. **Paralelização do Problema de Graph Matching para Grafos Exatos Utilizando CUDA**. 67 p. Trabalho de Conclusão do Curso (Ciência da Computação) — Universidade de Caxias do Sul, Caxias do Sul, 2020.

LEVINAS, M. **A Complete Introduction to GPU Programming with Practical Examples in CUDA and Python**. 2021. Disponível em: <https://www.cherryservers.com/blog/introduction-to-gpu-programming-with-cuda-and-python>. Acesso em: 13 jun. 2022.

LINDHOLM, E. *et al.* NVIDIA Tesla: A Unified Graphics and Computing Architecture. **IEEE Micro**, IEEE Computer Society Press, Washington, DC, USA, v. 28, n. 2, p. 39–55, mar 2008. ISSN 0272-1732. Disponível em: <https://doi.org/10.1109/MM.2008.31>.

LORIN, H. R. Review of "Highly Parallel Computing"by G. S. Almasi and A. Gottlieb, Benjamin-Cummings Publishers, Redwood City, CA, 1989. **IBM Syst. J.**, IBM Corp., Usa, v. 29, n. 1, p. 165–166, jan 1990. ISSN 0018-8670. Disponível em: <https://doi.org/10.1147/sj.291.0165>.

MASSAGO, S. **Introdução ao Fractal**. Tocantins: Universidade Federal de Tocantins, 2010. Disponível em: <https://www.dm.ufscar.br/profs/sadao/download/>. Acesso em: 13 jun. 2022.

MATPLOTLIB. **Matplotlib**. 2022. Disponível em: <https://matplotlib.org/>. Acesso em: 18 jun. 2022.

MOMMERSTEEG, V. Mandelbulb. In: INTERNATIONAL GAME ARCHITECTURE AND DESIGN, PROGRAMMING. Breda: NHTV University of Applied Sciences, 2014.

MOORE, G. E. Cramming More Components onto Integrated Circuits. **Electronics**, v. 38, n. 8, Abril 1965.

MORI, K. **Mandelbulb: Uma Visualização Tridimensional do Fractal de Mandelbrot**. Campinas: Unicamp, 2009. Disponível em: <https://www.blogs.unicamp.br/100nexos/2009/11/18/mandelbulb_uma_visualizao_trid/>. Acesso em: 5 jun. 2022.

NUMBA. **A high Performance Python compiler**. Numba, 2022. Disponível em: <https://numba.pydata.org/>. Acesso em: 10 jun. 2022.

____. **Numba Architecture**. Numba, 2022. Disponível em: <https://numba.pydata.org/numba-doc/latest/developer/architecture.html>. Acesso em: 5 jun. 2022.

____. **Numba for Cuda GPUs**. Numba, 2022. Disponível em: <https://numba.pydata.org/numba-doc/latest/cuda/index.html>. Acesso em: 5 jun. 2022.

NUMPY. NumPy, 2022. Disponível em: <https://numpy.org/>. Acesso em: 24 mar. 2022.

NVIDIA. **Overview**. NVIDIA, 2022. Disponível em: <https://nvidia.github.io/cuda-python/overview.html>.

PACHECO, P.; MALENSEK, M. **An Introduction to Parallel Programming**. 2. ed. Oxford, England: Morgan Kaufmann, 2021.

PALLESI, D. M. **Motivação do Estudo de Progressões Aritméticas e Geométricas Através da Geometria Fractal**. Paraná, 2007.

PARHAMI, B. **Introduction to Parallel Processing: Algorithms and Architectures**. New York: Kluwer Academic, 2002. ISBN 978-0306459702. Disponível em: <https://www.ece.ucsb.edu/~parhami/text/par/proc.htm>.

PAULA, T. M. R. d. S. Clayton Eugenio Santos de. Uma Abordagem da Geometria Fractal para o Ensino Médio. **Revista Eltrônica Paulista de Matemática**, v. 10, 2017.

PEITGEN, H.-O.; SAUPE, D. (Ed.). **The Science of Fractal Images**. New York, NY: Springer, 2011.

PILGRIM, I.; TAYLOR, R. Fractal Analysis of Time-Series Data Sets: Methods and Challenges. In: ____. [S.l.: s.n.], 2018. ISBN 978-1-78985-433-6.

PILLOW. Pillow, 2022. Disponível em: <https://pillow.readthedocs.io/en/stable/>. Acesso em: 24 mar. 2022.

PROENCA, C. B.; CONCI, A.; SEGENREICH, S. Comparação de Técnicas de Segmentação e de Dimensão Fractal visando a Detecção Automática de Falhas Têxteis. **Proceedings of VI Pan American Congress of Applied Mechanics**, v. 6, p. 381–384, jan 1999.

RAYPY. RayPY, 2022. Disponível em: <https://github.com/ryu577/pyray>. Acesso em: 18 jun. 2022.

REIS, M. V. dos. **Conjunto de Mandelbrot**. Goiás: Universidade Federal de Goiás, 2016.

SALLOW, A. Implementation and Analysis of Fractals Shapes using GPU-CUDA Model. **Academic Journal of Nawroz University**, v. 10, p. 1, 04 2021.

SANDERS, J.; KANDROT, E. **CUDA by Example: An Introduction to General-Purpose GPU Programming**. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 9780131387683 0131387685.

SANGIORGI, C.; COLLOP, A.; THOM, N. **A Non-destructive Impulse Hammer For Evaluating The Bond Between Asphalt Layers In A Road Pavement**. 2003. Disponível em: <https://www.ndt.net/article/ndtce03/papers/v095/v095.htm>.

SANTOS, B. P. dos; DOMINGUEZ, D. S. **Paradigmas de Processamento Paralelo na Resolução do Fractal de Mandelbrot**. Bahia: Universidade Estadual de Santa Cruz, 2012. Disponível em: <https://bps90.github.io/assets/files/2012-03-12-talk-erbase2012.pdf>.

SANTOS, B. P. dos; DOMINGUEZ, D. S.; ORELLANA, E. V. **O Problema do Fractal de Mandelbrot como Comparativo de Arquiteturas de Memória Compartilhada– GPU vs OpenMP**. Bahia: Universidade Estadual de Santa Cruz, 2011. Disponível em: <http://www.ifba.edu.br/anais_erad-ne2013/erad_ne_2011/PDF/PG/92823.pdf>.

SCHEPKE, C. Programação paralela em memória compartilhada e distribuída. In: XVIII ESCOLA REGIONAL DE ALTO DESEMPENHO, 2018, Porto Alegre. Porto Alegre: Erad, 2018. Disponível em: <http://www.inf.ufrgs.br/erad2018/downloads/minicursos/eradrs2018-mpi-openmp-slides.pdf>. Acesso em: 22 mai. 2022.

SEDREZ, M. **Conjunto de Cantor**. Santa Catarina: Universidade Federal de Santa Catarina, 2009. Disponível em: <http://www.avaad.ufsc.br/moodle/mod/hiperbook/view.php?id=2089&pagenum=7&target\%5Fnavigation\%5Fchapter=3713&show\%5Fnavigation=1>. Acesso em: 5 jun. 2022.

SEIBERT, S.; LAM, S. K. **Numba Tutorial for GTC 2017 Conference**. GitHub, 2017. Disponível em: <https://github.com/ContinuumIO/gtc2017-numba\#readme>. Acesso em: 7 jun. 2022.

SETH, S. **GPU Usage in Cryptocurrency Mining**. Investopedia, 2021. Disponível em: <https://www.investopedia.com/tech/gpu-cryptocurrency-mining/>. Acesso em: 7 jun. 2022.

SHILOV, A. **Graphics Cards Shipments Hit Three-Year High in Q3 2021: JPR**. Tom's Hardware, 2021. Disponível em: <https://www.tomshardware.com/news/jpr-q3-2021-desktop-discrete-gpu-shipments>. Acesso em: 9 jun. 2022.

STALLINGS, W. **Arquitetura e Organização de Computadores**. São Paulo: Pearson, 2017. v. 10.

THE Python Package Index. PyPi, 2022. Disponível em: <https://pypi.org/>. Acesso em: 24 mar. 2022.

TRACOLLI, M. **Parallel Generation of a Mandelbrot Set**. Virt&l-comm, 2016. Disponível em: <http://services.chm.unipg.it/ojs/index.php/virtlcomm/article/view/112>. Acesso em: 11 jun. 2022.

TSE, J. J. **Image Processing with CUDA**. 66 p. Dissertação (Mestrado em Ciência da Computação) — University of Nevada, 2012.

XU, C. Performance Optimization for DLA Model Based on GPU. **Spots–exemplified By Chengdu University Of Information Technology**, v. 2, n. 2, p. 118, 2014.

ZANOTTO, L.; FERREIRA, A.; MATSUMOTO, M. **Arquitetura e Programação de GPU NVIDIA**. Campinas: Universidade Estadual de Campinas, 2019. Disponível em: <https://ic.unicamp.br/~ducatte/mo401/1s2012/T2/G02-001963-023169-085937-t2.pdf>. Acesso em: 9 jun. 2022.

ZáVODSZKY, G. **Hemodynamic Investigation of Arteries Using the Lattice Boltzmann Method**. Tese (Doutorado) — University of Amsterdam, 04 2015.

ZHERU, Z.; HUAHAI, M.; CHENG, Q. Fractal Geometry of Element Distribution on Mineral Surfaces. **Mathematical Geology**, v. 33, n. 2, p. 217–228, 2001.