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

MATHEUS AUGUSTO TREGNAGO

IMPLEMENTAÇÃO PARALELA EM GPU PARA GERAÇÃO DE FRACTAIS DE CORAIS

BENTO GONÇALVES

2025

MATHEUS AUGUSTO TREGNAGO

IMPLEMENTAÇÃO PARALELA EM GPU PARA GERAÇÃO DE FRACTAIS DE CORAIS

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

BENTO GONÇALVES

2025

MATHEUS AUGUSTO TREGNAGO

IMPLEMENTAÇÃO PARALELA EM GPU PARA GERAÇÃO DE FRACTAIS DE CORAIS

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 em 24/11/2025

BANCA EXAMINADORA

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

Prof. Me. Samuel Francisco Ferrigo
Universidade de Caxias do Sul - UCS

Prof. Dr. Leonardo Pellizzoni
Universidade de Caxias do Sul - UCS

RESUMO

A geração de fractais é um processo computacionalmente intensivo, pois frequentemente são utilizados métodos iterativos que necessitam de um elevado número de iterações para formar estruturas complexas. Uma alternativa que tem sido amplamente utilizada para reduzir o tempo de execução do processo de geração de fractais consiste no uso de programação paralela, mais recentemente as unidades de processamento gráfico (GPUs). Apesar da vasta aplicação de fractais e do crescente uso de GPUs para sua geração, existem poucos de trabalhos dedicados à geração paralela de estruturas de corais. Portanto, este trabalho apresentou como objetivo desenvolver uma implementação paralela em GPU para a geração de fractais de corais em três dimensões. O método de geração adotado foi a Agregação Limitada por Difusão (DLA), um modelo tradicionalmente usado para simular estruturas tridimensionais encontradas na natureza. A implementação foi desenvolvida na linguagem de programação C++ e paralelizada através da plataforma CUDA da NVIDIA. Os resultados demonstraram que a versão paralela em GPU obteve desempenho superior em relação à implementação sequencial com um *speedup* de 40. Porém, observou-se uma queda e posterior estabilização do *speedup* a partir de 50.000 partículas, devido ao limite máximo de *threads* ativas simultaneamente na GPU.

Palavras-chave: Fractais de Corais, GPU, CUDA, Agregação Limitada por Difusão.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

CPU *Central Processing Unit*

CUDA *Compute Unified Device Architecture*

DDR *Double Data Rate*

DLA *Diffusion Limited Aggregation*

DRAM *Dynamic Random Access Memory*

GB *Gigabyte*

GHz *Gigahertz*

GPGPU *General-Purpose Computing on Graphics Processing Units*

GPU *Graphics Processing Unit*

KB *Kilobyte*

LSU *Load/Store Unit*

LTS *Long-Term Support*

MB *Megabyte*

MHz *Megahertz*

MIMD *Multiple Instruction, Multiple Data*

MISD *Multiple Instruction, Single Data*

NVMe *Non-Volatile Memory express*

OpenMP *Open Multi-Processing*

PCI-e *Peripheral Component Interconnect Express*

RAM *Random Access Memory*

SFU *Special Functions Unit*

SIMD *Single Instruction, Multiple Data*

SIMT *Single Instruction, Multiple Thread*

SISD *Single Instruction, Single Data*

SM *Streaming Multiprocessor*

SSD *Solid State Drive*

TB *Terabyte*

WSL *Windows Subsystem for Linux*

SUMÁRIO

1 INTRODUÇÃO

Um fractal é um objeto que mantém a forma invariável à medida que a escala de observação é alterada (ASSIS, 2008). A proposta de estudar a geometria fractal, conceito introduzido por Benoit Mandelbrot em 1975, surge do fato de que ela oferece uma descrição mais precisa da natureza em comparação com a geometria euclidiana. De fato, a geometria euclidiana não é capaz de descrever adequadamente várias das formas encontradas na natureza, como por exemplo, nuvens, montanhas, flores, árvores, etc (FUZZO, 2009). Assim, a geometria fractal está presente em estudos de diversas áreas da ciência e da natureza, como por exemplo, na trajetória de partículas (HANSEN *et al.*, 1997), ondas marítimas (YANG, 2014), compressão de arquivos (HAQUE *et al.*, 2014), entre outras.

Uma das áreas em que a geometria fractal é aplicada no estudo dos fenômenos da natureza é na análise da estrutura dos corais, onde os padrões fractais foram observados na estrutura em pequena escala de colônias de corais de águas rasas e na distribuição de corais de águas profundas (PURKIS, 2006). A geração dessas estruturas é utilizada para simular fenômenos naturais em diversas áreas, incluindo a simulação de um terreno em jogos e simulações computacionais, na qual softwares produzem imagens que procuram representar ou simular elementos da natureza (MENDONÇA, 2016).

O custo computacional para a geração de fractais é alto, visto que esses são formados a partir de um processo iterativo, que possibilita a formação de estruturas de elevada complexidade quando são empregadas muitas iterações (ASSIS, 2008). Dessa forma, a programação paralela torna-se uma abordagem atrativa para acelerar o processo de geração de fractais, pois, dependendo do tipo de fractal e do método utilizado, a geração de cada ponto pode ser feito de forma independente, permitindo assim uma distribuição entre os múltiplos núcleos de processamento (MAYFIELD *et al.*, 2016).

Entre as arquiteturas paralelas disponíveis, a unidade de processamento gráfico (GPU, do inglês, *Graphics Processing Unit*) tem se destacado como uma das alternativas mais eficientes. Atualmente, observa-se um aumento significativo na utilização desse tipo de arquitetura no desenvolvimento de aplicações paralelas (CORDEIRO; ZOLA, 2024; LEONARCZYK; GRIEBLER, 2023; FAé; GRIEBLER, 2024). Esse aumento deve-se ao elevado poder computacional dessa arquitetura em comparação com as unidades de processamento central (CPU, do inglês, *Central Processing Unit*). As GPUs possuem um grande número de núcleos de processamento, o que permite a execução simultânea de um número maior de tarefas, resultando em desempenho superior em comparação às CPUs (RAHMAD *et al.*, 2011).

Neste trabalho foi realizado um estudo sobre a utilização de GPUs para o desenvolvimento de uma aplicação paralela voltada para a geração de fractais de corais. A implementação

foi desenvolvida utilizando a linguagem de programação C++, escolhida por ser uma linguagem que possui um bom desempenho em tarefas de alto custo computacional. Além disso, a implementação foi paralelizada através do uso da plataforma CUDA (*Compute Unified Device Architecture*). Optou-se pela utilização da plataforma CUDA, pois essa foi desenvolvida especificamente para as GPUs da NVIDIA, que são as mais utilizadas atualmente (COYNE, 2025). Por fim, as implementações sequenciais e paralelas foram comparadas considerando o tempo total de execução.

1.1 OBJETIVOS

O objetivo geral deste trabalho consistiu no desenvolvimento de uma implementação paralela em GPU para a geração de fractais de corais, a fim de avaliar o ganho de desempenho. Para que atingir esse objetivo, os seguintes objetivos específicos foram realizados:

- Definir o fractal a ser implementado;
- Desenvolver uma implementação sequencial do fractal;
- Paralelizar a implementação para ser executada em GPUs;
- Comparar os tempos de execução das implementações sequencial e paralela.

1.2 ESTRUTURA DO TRABALHO

A estrutura deste trabalho está organizada da seguinte forma:

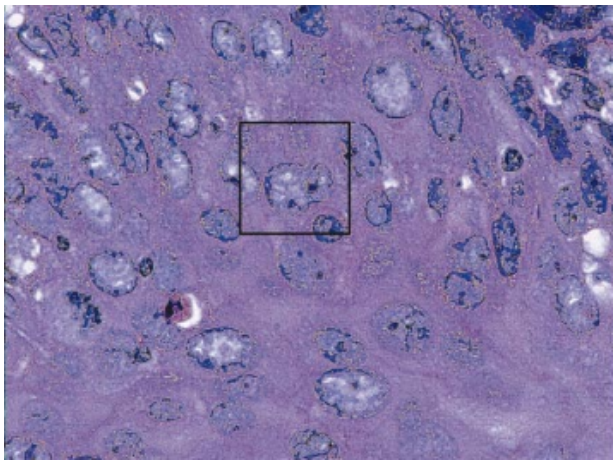
- Este capítulo apresentou uma visão geral do trabalho, destacando suas motivações e objetivos;
- No Capítulo 2 é feita uma conceituação sobre fractais. Além disso, são apresentados os trabalhos relacionados;
- No Capítulo 3 é realizada uma breve conceituação de arquiteturas paralelas. Esse capítulo detalha ainda a arquitetura das GPUs da empresa NVIDIA, que foi a arquitetura utilizada para o desenvolvimento deste trabalho;
- No Capítulo 4 é apresentado o fractal paralelizado neste trabalho. Além disso, são apresentadas a implementação sequencial e a implementação em GPU.
- No Capítulo 5 é descrita a avaliação da implementação paralela em GPU e o ambiente utilizado para os testes. Os resultados obtidos também são apresentados e discutidos.
- Por fim, no Capítulo 6 são apresentadas considerações finais e sugestões de trabalhos futuros.

2 GERAÇÃO DE FRACTAIS

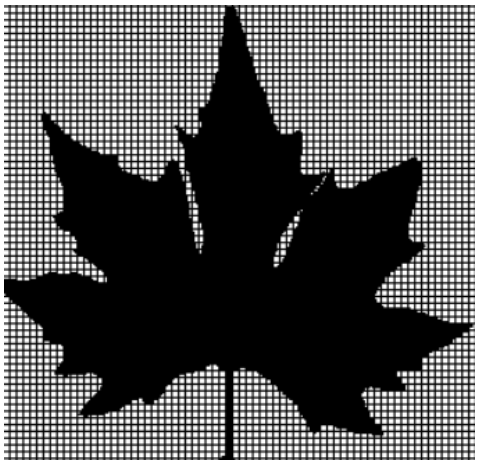
A geometria euclidiana é amplamente utilizada em estudos que possuem como objetivo compreender processos físicos e químicos (MEAKIN, 1990). No entanto, sua aplicação baseia-se no uso de superfícies planas, formas cristalinas poliédricas e partículas esféricas, o que limita sua utilização, principalmente à medida que a escala de análise se aproxima do comprimento atômico (MEAKIN, 1990). Desta forma, torna-se cada vez mais relevante a busca de uma descrição fractal para esses fenômenos, visto que um fractal possui um nível de detalhamento maior com a ampliação da estrutura (JURAEV; MAMMADZADA, 2024).

O termo fractal é utilizado para definir objetos que apresentam uma mesma estrutura em diferentes escalas. Esses objetos são formados por padrões que se repetem em versões menores, preservando uma semelhança com a estrutura original. O estudo dos fractais tem sido aplicado em diversas áreas do conhecimento. Na medicina, por exemplo, o estudo realizado por Sedivy *et al.* (1999) aborda o uso da geometria fractal como uma ferramenta para analisar núcleos celulares atípicos em lesões displásicas do colo do útero (Figura 1a). Neste trabalho, foi desenvolvido um método para avaliar a irregularidade dos núcleos em diferentes graus de neoplasia intraepitelial cervical. Na biologia, o estudo feito por Ayirli (2014) apresenta uma análise das dimensões fractais de folhas de diferentes espécies de plantas, visando propor uma nova metodologia para o estudo da taxonomia vegetal (Figura 1b).

Figura 1 – Exemplos de padrões fractais.



(a) Núcleo celular atípico.



(b) Planta fractal.

Fonte: Sedivy *et al.* (1999) e Ayirli (2014) respectivamente.

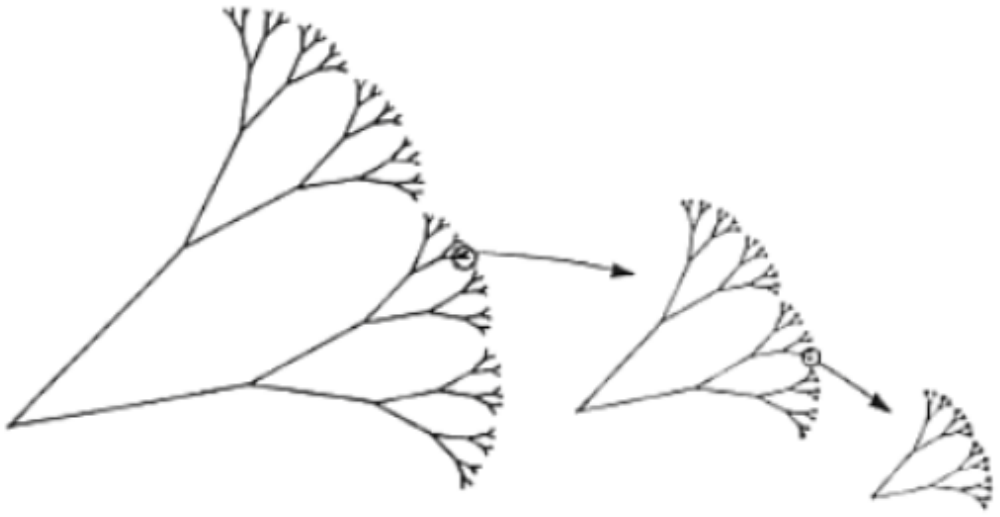
2.1 CARACTERÍSTICAS DOS FRACTAIS

O estudo dos fractais envolve a compreensão de suas propriedades e características matemáticas, especialmente no que se refere ao comportamento em relação ao comprimento, área e dimensões fractais (ASSIS, 2008). Neste sentido, os fractais caracterizam-se por três propriedades básicas: a auto-similaridade, que consiste na semelhança entre qualquer fragmento do fractal e a figura total; a complexidade infinita, que se refere ao processo iterativo infinito usado para a geração do mesmo; e a dimensão fractal, que é uma medida não exata que corresponde ao espaço ocupado pelo fractal.

2.1.1 Auto-similaridade

Uma das características fundamentais dos fractais é a auto-similaridade, onde as partes de um fractal se parecem com o todo, independentemente do nível de ampliação ou escala (SEI, 2003). Como pode ser observado na Figura 2, cada fragmento do fractal se apresenta como uma réplica do conjunto, mas em uma escala menor.

Figura 2 – Auto-similaridade



Fonte: Assis (2008).

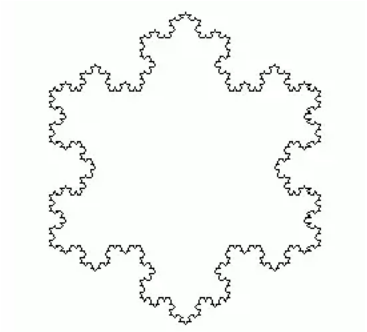
Segundo Fuzzo (2009), a auto-similaridade pode ser classificada em:

- Auto-similaridade exata: o fractal possui um formato idêntico, independente da escala de visualização. Um exemplo clássico de fractal dessa categoria é o fractal de Floco de Neve de Koch. Como pode ser observado Figura 3a, ao ampliar qualquer fragmento do fractal, é possível visualizar uma réplica perfeita do todo.
- Quase-auto-similaridade: o fractal apresenta uma aparência aproximadamente igual em diferentes escalas, embora não seja exatamente idêntico. Ou seja, em diferentes níveis de

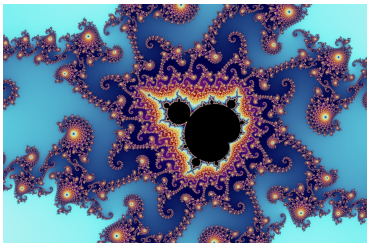
ampliação, pequenas diferenças podem ser observadas, o que tornam essa forma menos rígida que a exata. Essa forma de auto-similaridade pode ser observada no fractal de Mandelbrot (Figura 3b).

- Auto-similaridade estatística: é a forma mais sutil de auto-similaridade, onde o fractal não mantém uma aparência idêntica ou mesmo aproximada em diferentes escalas. No entanto, o fractal preserva medidas estatísticas de forma consistente, incluindo proporções e padrões de distribuição que se repetem em qualquer escala de visualização. Um exemplo de fractal com esse comportamento é o fractal de agregação por difusão limitada (DLA, do inglês *Diffusion Limited Aggregation*) (Figura 3c).

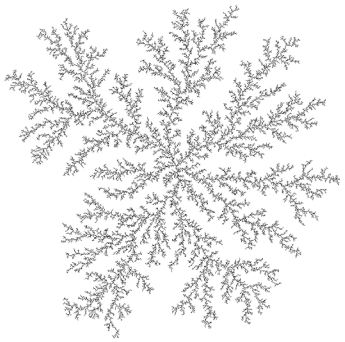
Figura 3 – Exemplos de fractais com diferentes tipos de autossimilaridade.



(a) Fractal Floco de Neve de Koch.



(b) Conjunto de Mandelbrot.



(c) Agregação por difusão limitada.

Fonte: Filho (2002), Mathigon (2023) e Bourke (1991), respectivamente.

2.1.2 Complexidade Infinita

A complexidade infinita de um fractal decorre do processo de geração do mesmo, que é recursivo. De fato, esse é construído por meio da repetição de um mesmo procedimento em diferentes escalas. Na construção iterativa de um fractal matematicamente definido, a recursão é aplicada infinitamente, resultando em uma estrutura que possui uma complexidade infinita (ASSIS, 2008).

2.1.3 Dimensão fractal

Na geometria euclidiana, os elementos fundamentais, como o ponto, a reta e o plano, apresentam dimensões topológicas de 0, 1 e 2, respectivamente (BICUDO, 2009). Assim, para determinar a dimensão topológica de um objeto, compara-se sua estrutura com as formas geométricas básicas (ponto, reta ou plano), a fim de identificar qual delas melhor representa a configuração fundamental do objeto (FUZZO, 2009).

As dimensões fractais (ou espaciais) referem-se ao espaço ocupado pelo objeto, independentemente da forma ou tamanho. Enquanto figuras da geometria euclidiana ocupam dimensões inteiras, os fractais podem apresentar dimensões fracionárias. Estas dimensões fracionárias podem ser calculadas a partir de dois métodos principais: dimensão por auto-similaridade e dimensão por contagem de caixas (ARSIE, 2008).

O método de cálculo de dimensão por auto-similaridade é utilizado quando o fractal apresenta uma propriedade de auto-similaridade exata. Neste caso, a dimensão D é definida com base na divisão de um objeto em N partes, cada uma reduzida por um fator R (ARSIE, 2008). Por exemplo, considerando-se uma reta, que possui dimensão $D = 1$, quando sua extensão é dividida, obtém-se $N = 2$, ou seja 2 cópias idênticas à original, cada uma com fator de redução $R = 1/2$. No caso de um quadrado com dimensão $D = 2$, quando os lados são divididos, obtém-se $N = 4$ cópias, cada uma com fator de redução $R = 1/2$. Da mesma forma, ao dividir as arestas de um cubo de dimensão $D = 3$, obtém-se $N = 8$ cópias, cada uma com fator de redução $R = 1/2$. Este padrão estabelece uma relação entre o número de cópias N, o fator de redução R e a dimensão D. Essa relação pode ser expressa através da Equação 2.1.

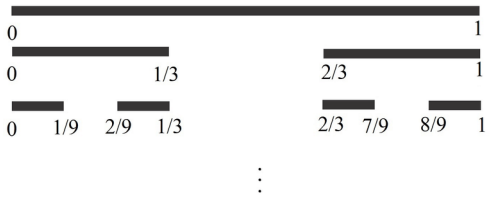
$$N = \left(\frac{1}{R}\right)^D \tag{2.1}$$

Isolando-se a variável de dimensão D obtém-se a Equação 2.2, que corresponde a dimensão fractal.

$$D = \frac{\log N}{\log\left(\frac{1}{R}\right)} \tag{2.2}$$

O método de cálculo de dimensão por auto-similaridade pode ser utilizado, por exemplo, para calcular a dimensão do fractal do Conjunto de Cantor (Figura 4). A lei da formação do Conjunto de Cantor consiste em dividir o intervalo $[0, 1]$ em três conjuntos iguais, $[0, \frac{1}{3}]$, $[\frac{1}{3}, \frac{2}{3}]$ e $[\frac{2}{3}, 1]$. Removendo-se o conjunto central ($[\frac{1}{3}, \frac{2}{3}]$), tem-se que os conjuntos restantes desse processo são $[0, \frac{1}{3}] \cup [\frac{2}{3}, 1]$. Esse processo é repetido k vezes para cada um dos conjuntos restantes, sendo que k tende ao infinito.

Figura 4 – Conjunto de Cantor



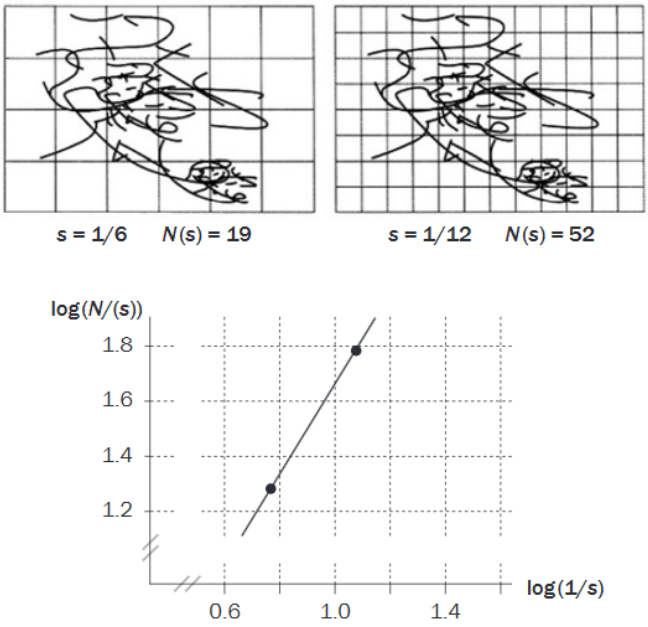
Fonte: Almeida e Santos (2017).

A partir da lei de formação do Conjunto de Cantor é possível calcular a dimensão fractal (ARSIE, 2008). A cada etapa do processo, divide-se o conjunto em 2, assim $N = 2$. Esse novo conjunto é obtido ao aplicar um fator de redução de $R = \frac{1}{3}$. A partir da Equação 2.2 obtêm-se uma dimensão D de 0,63, como pode ser observado na Equação 2.3.

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

Para realizar o cálculo de dimensões de fractais que não apresentam auto-similaridade exata, pode ser utilizado o método de dimensão por contagem de caixas, ou *box-counting* (COSTA, 2015). Este método consiste em cobrir o objeto com uma malha de quadrados com lados de tamanho s (Figura 5). Depois, é feita uma contagem de quantos quadrados possuem pelo menos uma parte do objeto a ser analisado. Essa contagem é representada por $N(s)$. Esse procedimento é repetido iterativamente utilizando malhas com quadrados de lados cada vez menores, obtendo-se os novos valores de N. Por fim, é gerado um gráfico logarítmico $\log N(s) \times \log \frac{1}{s}$, sendo traçada uma reta sobre os os pontos obtidos. A inclinação da reta obtida corresponde a dimensão do fractal.

Figura 5 – Exemplo *box-counting*



Fonte: Costa (2015).

No cálculo da dimensão por contagem de caixas, a cada ciclo de construção da malha, os lados dos quadrados da malha são reduzidos a um fator de $\frac{1}{2}$. Com isso, obtêm-se a sequência $N(2^{-k})$, onde k é o índice da etapa na iteração. A dimensão fractal do objeto é determinada pela inclinação da reta ajustada aos pontos do diagrama logarítmico, conforme expressa na Equação 2.4 (COSTA, 2015).

$$D = \frac{\log N(2^{-(k+1)}) - \log N(2^{-k})}{\log 2^{k+1} - \log 2^{k}} \tag{2.4}$$

A Figura 5 apresenta um exemplo do cálculo da dimensão fractal de um objeto que não possui uma autossimilaridade exata. Na parte superior da figura, os dois quadros ilustram a malhas formadas por quadrados de lado $s = \frac{1}{6}$ e $s = \frac{1}{12}$. A contagem de quadrados que possuem intersecção com uma parte do fractal são de $N(s) = 19$ para a malha de tamanho $s = \frac{1}{6}$ e $N(s) = 52$ para a malha de tamanho $s = \frac{1}{12}$. A inclinação da reta que une os dois pontos no gráfico $logN(s) \times log\frac{1}{s}$ é de $1,45$, conforme pode ser observado na Equação 2.5.

$$D_s = \frac{\log 52 - \log 19}{\log 12 - \log 6} \approx 1,45 \tag{2.5}$$

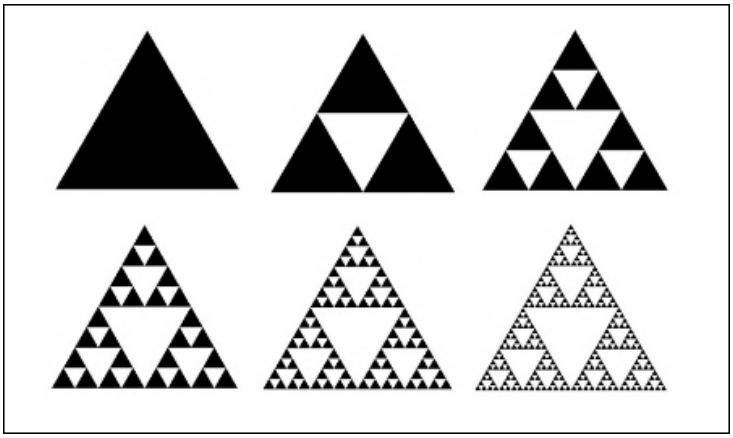
2.2 TIPOS DE FRACTAIS

De acordo com (FUZZO, 2009), os objetos fractais podem ser classificados em três categorias considerando o formato ou o modo de geração: sistema de funções iteradas ou geométricos; gerados por computadores ou tempo de escape; e randômicos ou aleatórios.

2.2.1 Fractais Geométricos

Os fractais de funções iteradas ou geométricos são gerados por meio da aplicação repetida de um conjunto finito de funções contrativas (operações de redução, deslocamento de figuras geométricas, etc). Um exemplo é o fractal do Triângulo de Sierpinski, que pode ser observado na Figura 6. Esse fractal é gerado a partir de um triângulo equilátero, onde é feita uma divisão em quatro outros triângulos, cujos vértices são os pontos médios dos lados do triângulo original. Em seguida, remove-se o triângulo central e repete-se esse processo sucessivamente nos triângulos restantes.

Figura 6 – Etapas da construção do triângulo de Sierpinski



Fonte: Adaptado de Santos *et al.* (2017).

2.2.2 Fractais de Tempo de Escape

Os fractais tempo de escape são objetos gerados a partir de iterações de equações no campo dos números complexos.[1] O processo de geração consiste em iterar uma função específica, onde diz-se que um ponto 'escapa' quando o valor da função associado a ele tende ao infinito. O tempo de escape é o número de iterações necessárias para determinar se o ponto escapará para o infinito. Um exemplo de fractal que se enquadra nessa categoria é o conjunto de Mandelbrot, que é formado pelo conjunto de pontos que não 'escapam' para o infinito.

O Conjunto de Mandelbrot é definido pela Equação 2.6, onde a iteração inicia-se com $z_0 = 0$ e c é um número complexo com o formato $c = x + yi$. Assim, para cada valor de c, a Equação 2.6 gera uma sequência de números complexos $(z_0, z_1, z_2, ...)$. Neste caso, o valor c pertence ao conjunto de Mandelbrot, se a sequência não tende ao infinito. Caso contrário o ponto não pertence ao conjunto de Mandelbrot.

$$z_{n+1} = z_n^2 + c \tag{2.6}$$

Por exemplo, para a constante $c = -2$, tem-se que a sequência é dada por:

$$\begin{aligned} z_0 &= 0 \\ z_1 &= 0^2 - 2 = -2 \\ z_2 &= (-2)^2 - 2 = 2 \\ z_3 &= (2)^2 - 2 = 2. \end{aligned}$$

Observa-se que para para qualquer valor $n \geq 2$, que o valor do termo da sequência é igual a 2 a partir da segunda iteração. Assim, o valor $c = -2$ pertence ao conjunto de Mandelbrot com um tempo de escape de 2. Já para a constante $c = 3$, tem-se que

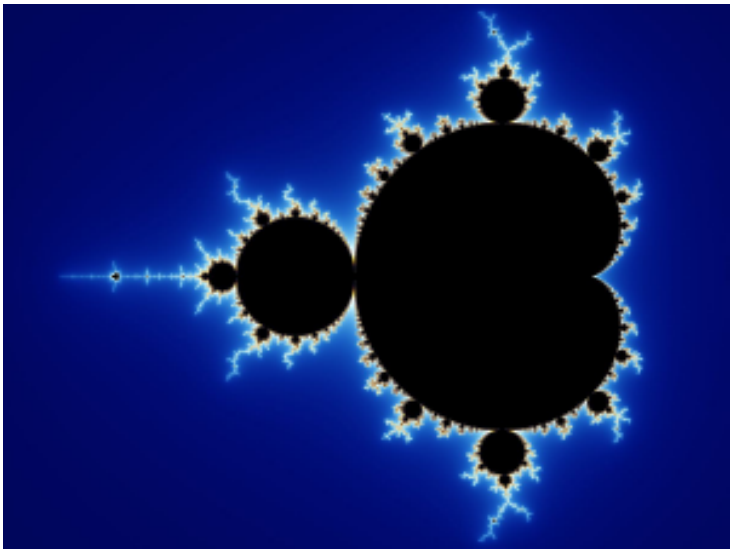
$$\begin{aligned} z_0 &= 0 \\ z_1 &= 0^2 + 3 = 3 \\ z_2 &= (3)^2 + 3 = 12 \\ z_3 &= (12)^2 + 3 = 147 \\ z_4 &= (147)^2 + 3 = 21612. \end{aligned}$$

Desta forma, o valor $c = 3$ não pertence ao conjunto de Mandelbrot, pois a sequência tende ao infinito.

[1] Espaço bidimensional em que cada ponto é representado por um número complexo da forma $a + bi$, em que a e b são números reais e i é a unidade imaginária

A Figura 7 apresenta o Conjunto de Mandelbrot, onde os pontos que percentem ao conjunto representados pela cor preta. Os pontos que não pertencem são coloridos de acordo com uma paleta de cores. A coloração dos pontos que não pertencem ao conjunto é determinada pelo número n de iterações necessárias até que a sequência $z_{k+1} = z_k^2 + c$ (iniciada em $z_0 = 0$) atinja um limite de fuga. Se o valor de z_n ultrapassar o limite de fuga, o cálculo é interrompido e o número de iterações n é registrado. A cor do pixel é então escolhida com base no valor de n. Se o limite de fuga não for alcançado após um número máximo de iterações, assume-se que o ponto c pertence ao Conjunto de Mandelbrot (GARCIA *et al.*, 2008).

Figura 7 – Conjunto de Mandelbrot



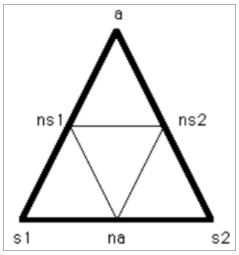
Fonte: Fractalize (2020).

2.2.3 Fractais Randômicos

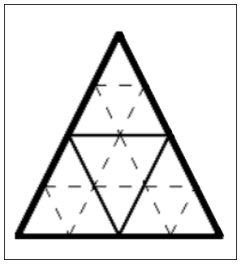
Os fractais randômicos ou aleatórios são gerados por processos estocásticos, introduzindo aleatoriedade em sua estrutura. Geralmente, esses fractais são utilizados para modelar fenômenos que exibem auto-similaridade estatística (FUZZO, 2009). Como exemplo, pode-se citar a Montanha Fractal, utilizada para geração de terrenos e cenários para jogos de computadores (MEIR; HALLER, 1991).

A geração da montanha fractal parte de um triângulo equilátero que, a cada iteração, é subdividido em quatro sub-triângulos congruentes (Figura 8a). Após, determinam-se os pontos médios das arestas (nsa, $ns1$ e $ns2$) e deslocam-se esses pontos verticalmente, de forma aleatória, dentro de um intervalo pré-definido. Esse processo forma quatro novos triângulos congruentes (Figura 8b), os quais serão submetidos ao mesmo procedimento. Esse procedimento repetido até atingir um número de iterações previamente definido, obtendo-se uma figura de uma montanha fractal, conforme exibido na (Figura 8 c).

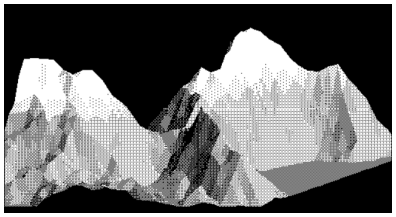
Figura 8 – Imagens da geração de uma montanha fractal.



(a) Triângulo equilátero inicial com vértices (s_1, s_2 e a) e com os pontos deslocados pelo algoritmo (ns_1, ns_2 e na).



(b) Triângulos formados a partir de duas iterações do algoritmo.



(c) Montanha fractal completa.

Fonte: Adaptado de Meir e Haller (1991).

2.3 TRABALHOS RELACIONADOS

Os fractais são amplamente empregados na avaliação de desempenho de ferramentas e ambientes de computação paralela, devido ao seu elevado custo computacional e a facilidade de paralelização. De fato, diversos estudos utilizam fractais, especialmente os do tipo escape, como os conjuntos de Mandelbrot e Julia, para a comparação de arquiteturas paralelas e ferramentas de programação. Além disso, observa-se um elevado número de trabalhos explorando a paralelização de fractais em arquiteturas de GPU.

No trabalho de Santos (2011), foram implementadas versões paralelas do fractal de Mandelbrot para ambientes de memória compartilhada, usando OpenMP, e para GPU, usando CUDA. Os testes foram realizados utilizando 4096 iterações para a geração dos fractais, com resoluções de 16384×16384. A implementação em OpenMP obteve um *speedup*[2] de $7,57$ e a implementação em CUDA obteve um *speedup* de $37,55$.

No trabalho realizado por Haque *et al.* (2014) foi desenvolvida uma implementação paralela em GPU de um algoritmo fractal para compressão de imagens médicas. A implementação realiza a compressão das imagens explorando a autossimilaridade das mesmas por meio de um sistema de funções iteradas. Para uma imagem de 512×512, o *speedup* obtido foi de 73, com uma taxa de compressão de 40%. Para imagens de 1024×1024, o *speedup* foi de 864, com uma taxa de compressão de 41%.

No trabalho desenvolvido por Sallow (2021), foram desenvolvidas versões paralelas em GPU dos fractais de Mandelbrot e de Julia. Para a geração dos fractais, foram realizadas 10000 iterações, sendo obtidos *speedups* de 347 para o fractal de Mandelbrot e 1139 para o de Julia.

[2] Medida de quanto um programa paralelo é mais rápido que a versão sequencial.

No trabalho de Siregar *et al.* (2022), também foi desenvolvida uma versão paralela para GPU do conjunto de Mandelbrot. Neste caso, foram realizados testes gerando o fractal com 10000 iterações e uma resolução de 512×512. A implementação para GPU obteve um *speedup* de aproximadamente 211.

No estudo realizado Duff (2023), foi desenvolvida uma versão paralela do conjunto de Mandelbrot para GPU. Os testes foram realizados em um *notebook* e no supercomputador da Universidade de *Cal Poly Humboldt*[3]. O fractal foi gerado utilizando 200 iterações, com uma resolução de 10000×10000. No supercomputador, foi obtido um *speedup* de 29 e no *notebook*, um *speedup* de 4.

Apesar do número considerável de trabalhos voltados à paralelização de fractais em GPUs, não foram observados na literatura estudos envolvendo a paralelização do processo de geração de corais utilizando fractais. No que se refere ao estudo dos corais, os conceitos de fractais são utilizados principalmente para compreender os processos envolvidos em sua formação.

Por exemplo, no trabalho desenvolvido por Martin-Garin *et al.* (2007), foi utilizado o método de contagem por caixas para calcular a dimensão fractal de diferentes espécies de corais. Neste trabalho, foi utilizado um método baseado nas dimensões fractais para caracterizar e distinguir as diferentes espécies de corais.

O estudo realizado por Purkis (2006) investiga as comunidades de corais e outros substratos marinhos no Golfo Pérsico. Usando imagens de satélite de alta resolução, os autores analisam a distribuição dos corais para verificar se estes seguem os padrões fractais. Neste caso, foi utilizado o método de contagem por caixas para calcular a dimensão fractal das estruturas. Os resultados mostraram que que as estruturas dos corais estudados seguem um comportamento fractal.

2.3.1 Análise Comparativa dos Trabalhos Relacionados

Uma análise comparativa dos trabalhos relacionados é apresentada no Quadro 1. É possível observar que os trabalhos presentes na literatura se concentram, na sua grande maioria, na geração de fractais de tempo de escape, como os conjuntos de Mandelbrot e Julia. Desta forma, verifica-se que poucos trabalhos abordam a geração de fractais que simulam elementos da natureza. Além disso, os estudos mais recentes utilizam a tecnologia CUDA para a paralelização do processo de geração de fractais (DUFF, 2023; SIREGAR *et al.*, 2022).

[3] Universidade pública localizada em Arcata, Califórnia.

Quadro 1 – Comparação entre trabalhos relacionados

Trabalho	**Objetivo**	**Tecnologias utilizadas**	**Resultados principais**
Santos (2011)	Implementar fractal de Mandelbrot em ambientes paralelos	OpenMP e CUDA	*speedup* de 7,57 com o OpenMP e 37,55 com o CUDA em resolução 16384×16384 com 4096 iterações
Haque *et al.* (2014)	Compressão de imagens médicas usando fractais	GPU com sistema de funções iteradas	*speedup* de 73 na resolução 512×512 e 864 na resolução 1024×1024, taxa de compressão 40%
Sallow (2021)	Geração dos fractais de Mandelbrot e Julia em GPU	CUDA	10000 iterações, *speedup* de 347 no conjunto de Mandelbrot e 1139 no conjunto de Julia
Siregar *et al.* (2022)	Versão paralela do conjunto de Mandelbrot em GPU	CUDA	10000 iterações, resolução 512×512, *speedup* de 211
Duff (2023)	Versão paralela do Mandelbrot em GPU	CUDA	Resolução 10000×10000, 200 iterações, *speedup* de 29 no supercomputador e 4 no notebook
Martin-Garin *et al.* (2007)	Calcular dimensão fractal de espécies de corais	Método de contagem por caixas	Diferenças fractais usadas para caracterizar espécies de corais
Purkis (2006)	Analisar distribuição de corais com imagens de satélite	*Box-counting* aplicado a imagens do Golfo Pérsico	Estruturas de corais apresentaram comportamento fractal
Zsaki (2016)	Paralelizar o método DLA para geração de estruturas 3D	*OpenCL* em CPU e GPU	Grade 768×768×768, *speedup* de 8 na CPU e 55 na GPU

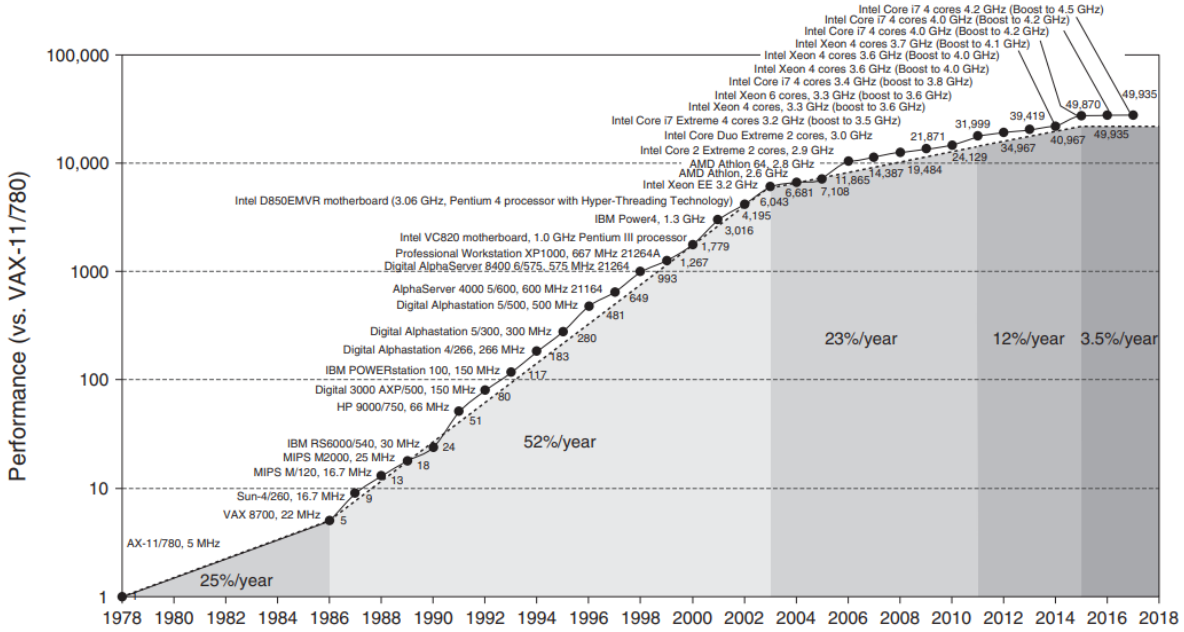
Fonte: O Autor (2025).

Diante da escassez de estudos voltados à geração de fractais que modelem elementos da natureza, especialmente no que se refere à geração de fractais de corais, optou-se pelo desenvolvimento de uma versão paralela desse algoritmo em GPU. Para o desenvolvimento deste trabalho, optou-se pela utilização do método DLA (do inglês *Diffusion Limited Aggregation*), visto que esse método é frequentemente utilizado para a geração de fenômenos e estruturas existentes na natureza. Trabalhos desenvolvidos mostraram a viabilidade da paralelização do método DLA em GPUs. De fato, no trabalho desenvolvido por Zsaki (2016), o método DLA foi paralelizado para a geração de estruturas 3D. O algoritmo foi paralelizado em CPUs e GPUs usando a plataforma *OpenCL*. As simulações foram realizadas utilizando uma grade de $768 \times 768 \times 768$. Os resultados obtidos foram de uma *speedup* de aproximadamente 8 em CPUs e de de aproximadamente 55 em GPUs.

3 UNIDADES DE PROCESSAMENTO GRÁFICO

Nos últimos anos os fabricantes têm encontrado dificuldades em aumentar o poder computacional dos processadores, principalmente devido a questões de consumo de energia e dissipação de calor (KIRK; HWU, 2010). Essa desaceleração pode ser visualizada na Figura 9, que mostra o crescimento do desempenho dos processadores ao longo de 40 anos, usando como referência o processador VAX 11/780.

Figura 9 – Evolução das CPUs



Fonte: Hennessy e Patterson (2017).

Uma alternativa para contornar esse problema foi o desenvolvimento de arquiteturas paralelas, onde diversas unidades de processamento trabalham em conjunto para acelerar a execução de um problema computacional (STALLINGS, 2017; KIRK; HWU, 2010). Existem diversos tipos de arquiteturas paralelas, que tradicionalmente são classificadas utilizando Taxonomia de Flynn (TANENBAUM, 2013; HENNESSY; PATTERSON, 2017).

A Taxonomia de Flynn classifica as arquiteturas computacionais com base em seu grau de paralelismo, distinguindo entre modelos paralelos e não paralelos (FLYNN, 1966). Nesta, as arquiteturas computacionais são classificadas de acordo com o número de fluxos de instruções e do número de fluxos de dados. As classificações presentes na taxonomia de Flynn são:

- *Single Instruction, Single Data* (SISD): é a arquitetura de um computador sequencial clássico. Esta arquitetura caracteriza-se por apresentar apenas um fluxo de instruções que opera sobre um único fluxo de dados. O exemplo mais comum para esta categoria são os computadores com apenas um núcleo de processamento (*single-core*).

- *Single Instruction, Multiple Data* (SIMD): um único fluxo de instruções é executado sobre múltiplos fluxos de dados. Neste caso, uma única unidade de controle emite uma instrução, e várias unidades de processamento executam essa instrução em em diferentes conjuntos de dados. Exemplos dessa arquitetura incluem os supercomputadores vetoriais e matriciais e, mais recentemente, as GPUs.

- *Multiple Instruction, Single Data* (MISD): nesta arquitetura, múltiplos fluxos de instruções operariam sobre um único fluxo de dados. Não existem exemplos práticos dessa arquitetura, embora alguns autores sugerem que máquinas com *pipeline*[1] poderiam se enquadrar nessa classificação.

- *Multiple Instruction, Multiple Data* (MIMD): é considerada a categoria mais abrangente de arquiteturas paralelas. A arquitetura MIMD é caracterizada pela execução de múltiplos fluxos de instruções sobre múltiplos fluxos de dados. Nela, várias unidades de processamento independentes operam sobre conjuntos de dados distintos. Os principais exemplos dessa arquitetura são os computadores com processadores de múltiplos núcleos (*multi-core*), multiprocessadores e os multicomputadores.

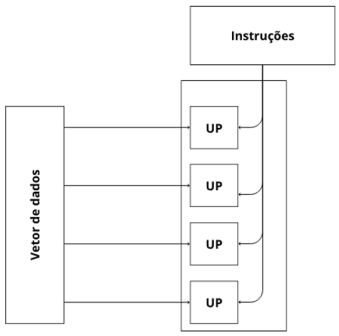
A arquitetura utilizada para a paralelização da geração de fractais de corais são as GPUs, que se encontram na categoria SIMD. Sendo assim, a arquitetura SIMD será apresentada com maiores detalhes.

3.1 ARQUITETURA SIMD

A arquitetura SIMD caracteriza-se por uma organização computacional onde uma única unidade de controle gerencia múltiplos núcleos de processamento (Figura 10). Essa unidade de controle decodifica um único fluxo de instruções e o transmite para todos os processadores, que executam simultaneamente a mesma instrução, cada um operando sobre um conjunto distinto de dados. A comunicação e a troca de dados entre os processadores é realizada tradicionalmente através de uma memória compartilhada (HENNESSY; PATTERSON, 2017).

[1] Uma instrução é dividida em estágios e processada sequencialmente por diferentes unidades

Figura 10 – Arquitetura SIMD



Fonte: O Autor (2025).

De forma geral, as arquiteturas do tipo SIMD podem ser classificadas em dois principais tipos: os processadores matriciais e os vetoriais. Um processador vetorial é uma unidade equipada com componentes especializados para operar sobre conjuntos de dados por meio de uma única instrução. Trata-se de um modelo autônomo, pois possui sua própria unidade de controle, ou seja, não necessita de outro dispositivo para gerenciar as operações. Além disso, os processadores vetoriais podem efetuar operações escalares, ao contrário dos processadores matriciais que dependem de outra unidade de controle para efetuar essas operações (HENNESSY; PATTERSON, 2017).

Um processador matricial é uma unidade de processamento que não conta com uma unidade de controle própria. A estrutura é composta por múltiplas unidades de processamento simples com memória local, interligadas por uma rede de interconexão para a troca de informações. Assim, esse processador atua como um auxiliar em uma estrutura que envolve duas partes: o *host* e o *device*. O *host* é o processador principal, que possui a unidade de controle e gerencia toda a operação. Já o *device* é o processador matricial que executa as tarefas de computação intensiva (ABD-EL-BARR; EL-REWINI, 2005). As GPUs são um exemplo de processadores matriciais, onde a CPU age como *host* e a GPU como *device*.

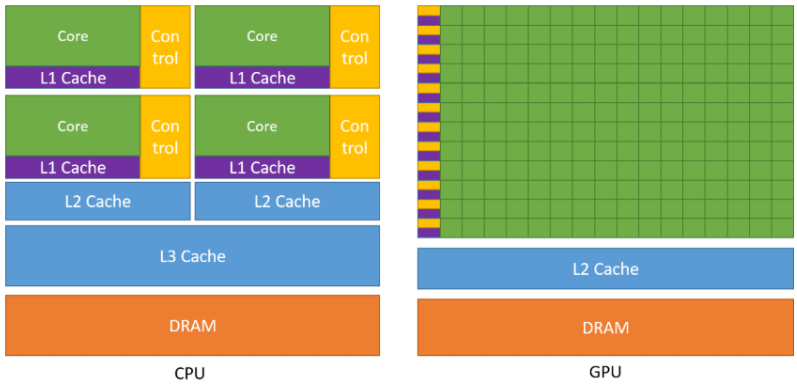
3.2 UNIDADES DE PROCESSAMENTO GRÁFICO

Embora a arquitetura SIMD tenha sido desenvolvida para ser usada em supercomputadores, ela atualmente está presente em diversos computadores pessoais a partir das unidades de processamento gráfico. Inicialmente, as GPUs foram projetadas para aplicações de processamento e renderização gráfica (THOMAS; DARUWALA, 2014). Contudo, devido a sua capacidade de processar grandes volumes de dados, essas tornaram-se atrativas para uma ampla gama de aplicações. Com isso, surgiram as GPUs de propósito geral, conhecidas como GPGPUs (do inglês *General-Purpose Computing on Graphics Processing Units*) (KIRK; HWU, 2010). Atualmente, as GPGPUs são utilizadas em diversas áreas do conhecimento que demandam de alto poder de processamento, como por exemplo, simulação de fluidos (HÖLSCHER *et al.*, 2025), sis-

temas distribuídos (YAO; HU; HOU, 2025; RYU; BYEON; KIM, 2025), entre outras (OWENS *et al.*, 2008).

O desempenho superior das GPUs em relação às CPUs decorre das diferenças entre as duas arquiteturas (Figura 11). As CPUs são projetadas com poucos núcleos, porém mais complexos, com lógicas de controle sofisticadas e hierarquias de *cache* elaboradas, sendo otimizadas para a execução de tarefas sequenciais ou com paralelismo limitado (HENNESSY; PATTERSON, 2017). Por outro lado, as GPUs são constituídas por milhares de núcleos, individualmente mais simples, mas otimizados para a execução simultânea de múltiplas instruções sobre dados diferentes (NVIDIA, 2025a).

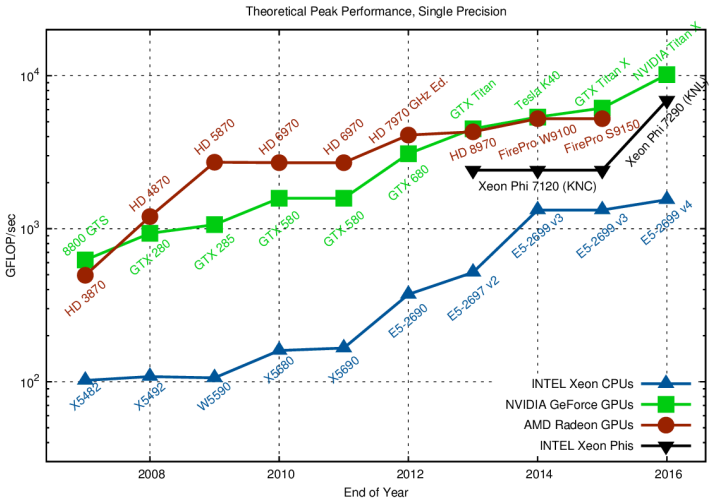
Figura 11 – Arquitetura CPU × GPU



Fonte: NVIDIA (2025a).

Essas diferenças entre as arquiteturas das CPUs e GPUs, contribuíram para que as GPUs, com sua arquitetura voltada ao paralelismo massivo, apresentassem um poder de processamento significativamente maior, conforme pode ser observado na Figura 12.

Figura 12 – Comparativo de desempenho entre GPUs e CPUs



Fonte: Rupp (2016).

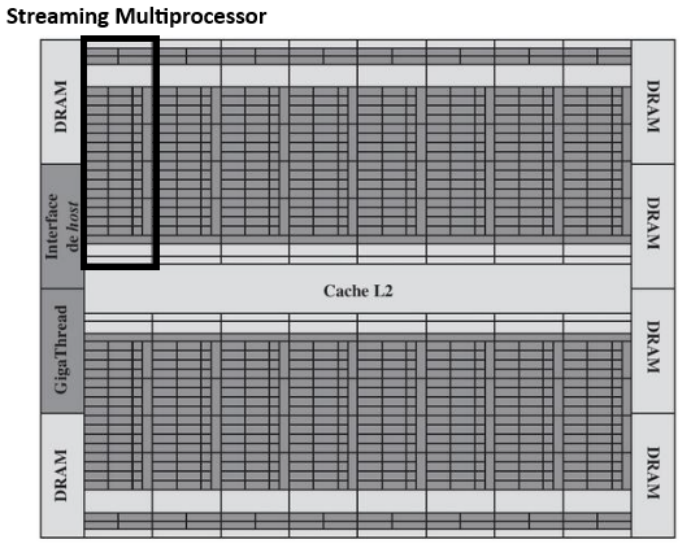
3.3 ARQUITETURA DA UNIDADE DE PROCESSAMENTO GRÁFICO

A GPU utilizada no desenvolvimento deste trabalho é da fabricante NVIDIA. Por esse motivo, optou-se por apresentar a arquitetura de uma GPU tomando como referência a arquitetura das GPUs desenvolvida por essa empresa. Mais especificamente, optou-se por apresentar a arquitetura Fermi da NVIDIA (STALLINGS, 2017).

A arquitetura das GPUs da NVIDIA é composta por um conjunto de SMs (*Streaming Multiprocessor*), que possibilitam a execução simultânea de várias *threads*[2]. Nesta arquitetura, as *threads* são organizadas em conjuntos chamados de *warp* (conjunto de 32 *threads*). Assim, todas as *threads* de um *warp* são executadas simultaneamente por um SM. A execução dessas *threads* segue o modelo SIMT (do inglês *Single Instruction, Multiple Thread*), onde todas as *threads* de um *warp* executam simultaneamente uma mesma instrução.

Na Figura 13, tem-se uma ilustração da arquitetura Fermi da NVIDIA. Essa possui 16 SMs organizados em dois grupos de 8 SMs cada. Por sua vez, cada SM contém 32 núcleos CUDA, totalizando 512 núcleos. Os núcleos CUDA são as unidades de processamento responsáveis pela execução de operações com número inteiros e de ponto flutuante. Para otimizar o acesso aos dados, uma memória *cache L2* centralizada atende a ambos os grupos de SMs. A arquitetura também integra seis unidades de memória DRAM e uma interface PCI-e que realiza a comunicação com a CPU. Essa interface contém uma memória de acesso direto para a transferência de dados entre a memória do *host* e a memória da GPU (KIRK; HWU, 2010). Por fim, tem-se o escalonador global *GigaThread* que é responsável pela distribuição e gerenciamento das *threads* entre os diferentes SMs (STALLINGS, 2017).

Figura 13 – Arquitetura Fermi NVIDIA

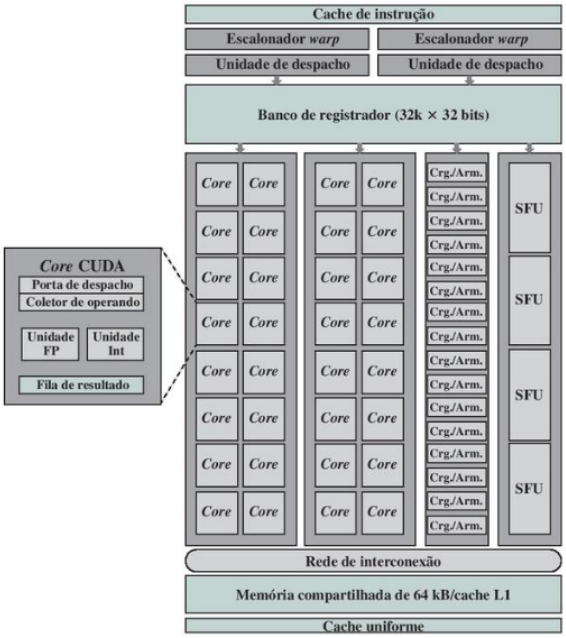


Fonte: Adaptado de Stallings (2017).

[2] Uma *thread* é um fluxo de execução de intruções.

Como pode ser observado na Figura 14, cada SMs é composto por 32 núcleos CUDA. Cada núcleo CUDA é projetado com duas unidades, sendo uma dedicada ao processamento de números inteiros e outra de números em ponto flutuante. No entanto, apenas uma das unidades pode ser utilizada por ciclo de *clock*. Ademais, uma operação em ponto flutuante de precisão simples pode ser executada por uma única unidade de ponto flutuante. Já uma operação de precisão dupla exige o uso combinado de dois núcleos CUDA. Desta forma, *threads* com cálculos de precisão dupla necessitam de mais tempo para serem executadas em comparação com aquelas que utilizam operações de precisão simples (STALLINGS, 2017).

Figura 14 – Arquitetura de um SM



Fonte: Stallings (2017).

Além disso, cada SM, possui ainda um conjunto de 4 unidades de funções especiais (SFU, do inglês *Special Functions Unit*). Esses componentes são projetados para executar operações matemáticas em apenas um ciclo de *clock*. Essas unidades são capazes de executar operações matemáticas especiais como seno, cosseno, exponencial, logaritmos, potências e raízes (STALLINGS, 2017).

Os escalonadores de *warp* e as portas de despacho operam em conjunto com o escalonador *GigaThread*. O escalonador *GigaThread* realiza a distribuição das *threads* para os diferentes SMs, sendo que cada SM divide as *threads* em *warps* de 32 *threads*. Neste caso, cada *warp* apresenta seu próprio escalonador. Todas as *threads* que compõem um *warp* iniciam executando uma mesma instrução, porém cada *thread* possui seu próprio contador de programa e um conjunto de registradores. Assim, cada *thread* pode criar ramificações de forma independente. Porém, se as *threads* de um *warp* divergirem devido a uma ramificação, o *warp* executa serialmente cada caminho da ramificação, desativando as demais *threads* que não pertencem ao caminho ativo (NVIDIA, 2025a).

As unidades de carga/armazenamento (LSU, do inglês *Load/Store Unit*)) são responsáveis por gerenciar o acesso à memória em cada SM. A principal função de cada LSU é calcular o endereço de origem ou destino de uma instrução. Esses endereços podem referenciar a memória *cache* do SM ou a memória DRAM, para a comunicação com a CPU (STALLINGS, 2017).

3.3.1 Hierarquia de Memória de uma GPU

Uma GPU possui diferentes níveis memórias, onde cada nível de memória possui seu propósito e características. Normalmente, esses níveis são divididos em dois tipos: memórias que estão dentro de um SM *(on-chip)* e memórias que estão fora de um SM *(off-chip)*. De acordo com Stallings (2017) e Hennessy e Patterson (2017), os diferentes níveis de memórias presentes em uma GPU são:

- Registradores: memória *on-chip* de mais alta velocidade. Esses são utilizados para o armazenamento de dados privados de cada *thread*. Assim, cada *thread* pode acessar apenas o seu conjunto de registradores.
- Memória Compartilhada: memória *on-chip*, sendo utilizada para o compartilhamento de dados entre todas as *threads* de um mesmo *warp*.
- Memórias Locais: memória *off-chip* que é utilizada para o armazenamento de dados cujo tamanho excede a capacidade dos registradores de cada *thread*. Essa memória é privada para cada *thread*.
- Memória Global: memória *off-chip* que é compartilhada por todas as *threads* de todos os *warps*. Essa memória também é acessível pela *CPU* do *host*.
- Memória de Constante: memória *off-chip*, mas que possui uma *cache* dedicada dentro de cada SM. Essa é uma memória só para leitura (*read-only*) que é utilizada para o compartilhamento de dados entre todas as *threads* de todos os *warps*.
- Memória de Textura: esta memória *off-chip* que é otimizada para operações de leitura e utiliza o mesmo espaço físico da memória global. Essa é otimizada para o acesso a estruturas de dados unidimensionais, bidimensionais ou tridimensionais.

O Quadro 2 apresenta um resumo dos diferentes níveis de memórias presentes na arquitetura de uma GPU.

Quadro 2 – Hierarquia de memória em uma GPU

Tipo de memória	**Tempo de acesso relativo**	**Localização**	**Tipo de acesso**	**Escopo**
Registradores	O mais rápido	*On-chip*	Leitura/Escrita	*Thread* única
Compartilhada	Rápido	*On-chip*	Leitura/Escrita	Todas as *threads* em um *warp*
Local	Entre 100 e 150 vezes mais lento que a compartilhada e os registradores	*Off-chip*	Leitura/Escrita	*Thread* única
Global	Entre 100 e 150 vezes mais lento que a compartilhada e os registradores	*Off-chip*	Leitura/Escrita	Todas as *threads* e *host*
Constante	Entre 100 e 150 vezes mais lento que a compartilhada e os registradores	*Off-chip*	Leitura	Todas as *threads* e *host*
Textura	Entre 100 e 150 vezes mais lento que a compartilhada e os registradores	*Off-chip*	Leitura	Todas as *threads* e *host*

Fonte: Adaptado de Stallings (2017).

3.3.2 Compute Unified Device Architecture (CUDA)

A *Compute Unified Device Architecture* (CUDA) é uma plataforma de computação criada pela NVIDIA em 2006 para facilitar o desenvolvimento de aplicações para as GPUs desenvolvidas pela empresa (NVIDIA, 2025a). Um programa em CUDA é dividido em trechos de códigos que são executados pelo *host* (CPU) ou pelo *device* (GPU). Os trechos de código sequenciais são executados pelo *host* e os trechos de código que necessitam de paralelização são executados no *device*. Um programa CUDA é escrito em um único código fonte, que engloba tanto as instruções para o *host* quanto para o *device* (KIRK; HWU, 2010).

Dentro de um programa em CUDA, as funções a serem executadas na GPU são denominadas *kernels*. Um *kernel* é definido a partir da declaração `__global__`, conforme pode ser visto na primeira linha do Algoritmo 1, que define um *kernel* que realiza a soma de dois vetores e armazena o resultado em um terceiro.

Algoritmo 1 – Soma de vetores usando um *kernel*

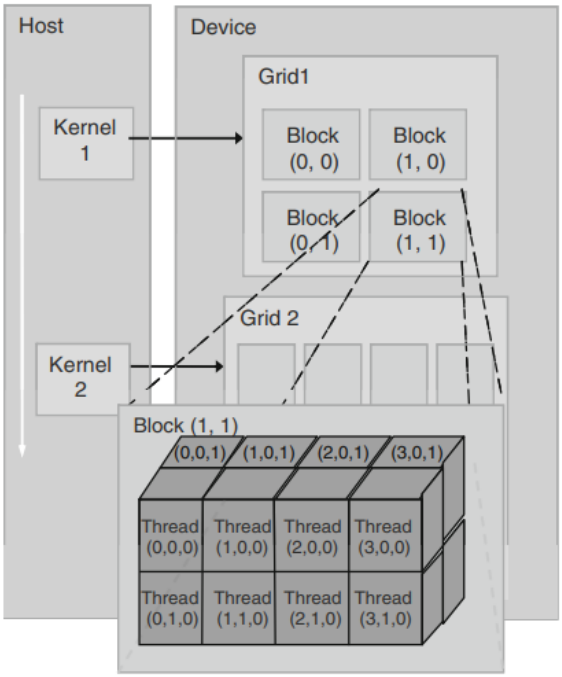
```
1  __global__ void VecAdd(float* A, float* B, float* C){
2      int i = threadIdx.x;
3      C[i] = A[i] + B[i];
4  }
5
6  int main(){
7      ...
8      // Chamada do kernel com N blocos e N threads por bloco
9      VecAdd<<<NBLOCKS, NTHREADS>>>(A, B, C);
10     ...
11 }
```

Fonte: NVIDIA (2025a).

No momento da chamada de um *kernel* é necessário definir o número de *threads* que serão executadas. Dentro de um programa CUDA, as *threads* podem ser agrupadas em blocos unidimensionais, bidimensionais ou tridimensionais. A invocação do *kernel* é feita a partir da sintaxe $<<< BLOCKS, THREADS >>>$, onde o primeiro parâmetro representa o número de blocos de *threads* e o segundo representa a quantidade de *threads* de cada bloco. Por exemplo, caso a inicialização do *kernel* for com feita com 4 blocos e 4 *threads* por bloco, o *kernel* será executado por um total de 16 *threads*. O Algoritmo 1 exemplifica a chamada do *kernel VecAdd* na linha 9.

Os blocos de *threads* podem ser agrupados em um *grid*. Um exemplo de *grid* pode ser visualizado na Figura 15, que apresenta dois *kernels*, cada um sendo executado por um *grid*. Esse *grids*, por sua vez, possuem uma estrutura bidimensional de blocos, onde cada bloco é representado por uma estrutura tridimensional de *threads*.

Figura 15 – Representação de *Grid* de *threads* em CUDA



Fonte: Kirk e Hwu (2010).

Em CUDA, existem variáveis embutidas que só podem ser utilizadas dentro de um *kernel*. Essas variáveis são frequentemente utilizadas para a divisão do processamento entre as *threads*, pois possibilitam que as *threads* e blocos sejam diferenciados. Ou seja, a composição dessas variáveis embutidas funcionam como um identificador único (NVIDIA, 2025a). As variáveis embutidas permitem que divisão seja realizada a nível de *thread* ou de bloco.

A posição de uma *thread* em um bloco pode ser acessado através das variáveis embutidas `threadId.x`, `threadId.y` e `threadId.z`. No caso onde o *kernel* é inicializado contendo blocos de apenas uma dimensão, a variável `threadId.x` irá retornar a posição de uma *thread* dentro do referido bloco. Caso seja criado um bloco com duas dimensões, a variável `threadId.x` irá retornar a posição de uma *thread* em relação ao eixo `x`, e variável `threadId.y` a posição de uma *thread* em relação ao eixo `y`. A mesma lógica aplica-se à organização de blocos em três dimensões. No Algoritmo 2, tem-se um exemplo de uso das variáveis `threadId.x` (linha 5) e `threadId.y` (linha 6) para o acesso o acesso aos elementos de uma matriz.

Algoritmo 2 – Soma de matrizes usando um *kernel*

```
1 __global__ void MatAdd(float A[N][N], float B[N][N],
2                         float C[N][N])
3 {
4     int i = threadIdx.x;
5     int j = threadIdx.y;
6     C[i][j] = A[i][j] + B[i][j];
7 }
```

Fonte: NVIDIA (2025a).

A variável `blockDim` pode ser utilizada para obter as dimensões do bloco de *threads*. No caso de um bloco criado com uma única dimensão, pode-se obter a dimensão do mesmo através da variável `blockDim.x`. No caso de um bloco criado em duas dimensões obtém-se as dimensões do bloco através das variáveis `blockDim.x` e `blockDim.y`. Já para blocos de três dimensões são utilizadas as variáveis `blockDim.x`, `blockDim.y` e `blockDim.z`. O Algoritmo 3 exemplifica o uso das variáveis `blockDim.x` e `blockDim.y` para a obtenção das dimensões de um bloco bidimensional.

Algoritmo 3 – Obtenção das dimensões de um bloco de *threads*

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

Fonte: NVIDIA (2025a).

A GPU não pode acessar os dados que encontram-se na memória do *host*. Para tanto, é necessário primeiro alocar um espaço na memória da *GPU* e, em seguida, transferir os dados. A alocação na memória da GPU é realizada através da função `cudaMalloc` e a transferência dos dados é realizada através da função `cudaMemcpy` (NVIDIA, 2025a; KIRK; HWU,

2010). O Algoritmo 4 exemplifica o uso das funções `cudaMalloc` e `cudaMemcpy`. A função `cudaMalloc` é utilizada para a alocação dos vetores `hA`, `hB` e `hC` (linhas 8, 9 e 11). A transferência dos vetores para a memória do *device* é realizada através da função `cudaMemcpy` (linhas 15 e 16). O parâmetro `cudaMemcpyHostToDevice`, indica que a transferência é feita da memória do *host* (CPU) para o *device* (GPU). Após o término da execução do *kernel*, a função `cudaMemcpy` é utilizada para a transferência dos resultado da memória do *device* para a memória do *host* (linha 22). Essa direção de transferência é definida através do parâmetro`cudaMemcpyDeviceToHost`. Por fim, o `cudaFree` (linhas 24, 25 e 26) é utilizado para liberar a memória alocada no *device* das variáveis `dA`, `dB` e `dC`.

Algoritmo 4 – Alocação de memória na GPU

```
1   // Aloca vetores de dados hA and hB na memoria do host
2   float* hA = (float*)malloc(size);
3   float* hB = (float*)malloc(size);
4   float* hC = (float*)malloc(size);
5
6   // Aloca vetores na memoria do device
7   float* dA;
8   cudaMalloc(&dA, size);
9   float* dB;
10  cudaMalloc(&dB, size);
11  float* dC;
12  cudaMalloc(&dC, size);
13
14  // Copia os vetores da memoria do host para a memoria do device
15  cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);
16  cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);
17
18  ... // Executa um kernel na GPU
19
20  // Copia o resultado da memoria do device para a memoria do host
21  // hC contem o resultado na memoria do host
22  cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);
23  cudaFree(dA);
24  cudaFree(dB);
25  cudaFree(dC);
26  ...
```

Fonte: Adaptado de NVIDIA (2025a).

Uma vez que os dados estão na memória GPU, a plataforma CUDA oferece um conjunto de qualificadores para gerenciar o uso da hierarquia de memória da GPU. A escolha adequada desses qualificadores é fundamental, uma vez que eles exercem influência significativa sobre o desempenho do programa (KIRK; HWU, 2010). O acesso a essas memórias é controlado por através dos qualificadores: `__device__`, `__constant__` e `__shared__`. O qualificador

__device__ especifica que a variável é alocada na memória global da GPU, ficando acessível à todas as *threads* e à CPU. Por outro lado, o qualificador __constant__ aloca a variável na memória constante da GPU. A memória constante é acessível apenas no modo de leitura pelas *threads* da GPU, sendo modificável exclusivamente pela CPU antes da criação do *kernel*. Por sua vez, as variáveis __shared__ são alocadas na memória compartilhada de um SM, sendo acessíveis durante a execução do *kernel* por todas as *threads* do bloco (KIRK; HWU, 2010). No Algoritmo 5 tem-se um exemplo da utilização dos qualificadores para o acesso aos diferentes níveis 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: Adaptado de NVIDIA (2025a).

4 IMPLEMENTAÇÃO DESENVOLVIDA

Este trabalho apresentou como principal objetivo desenvolver uma implementação paralela em GPU para a geração de estruturas de corais em três dimensões. Para isso, foi utilizado o algoritmo Agregação Limitada por Difusão (DLA, do inglês *Diffusion Limited Aggregation*) (BRAGA; RIBEIRO, 2011). Neste capítulo, inicialmente é apresentado o método DLA. Em seguida, é descrita sua implementação sequencial e, por fim, é descrita a implementação paralela utilizando a plataforma CUDA.

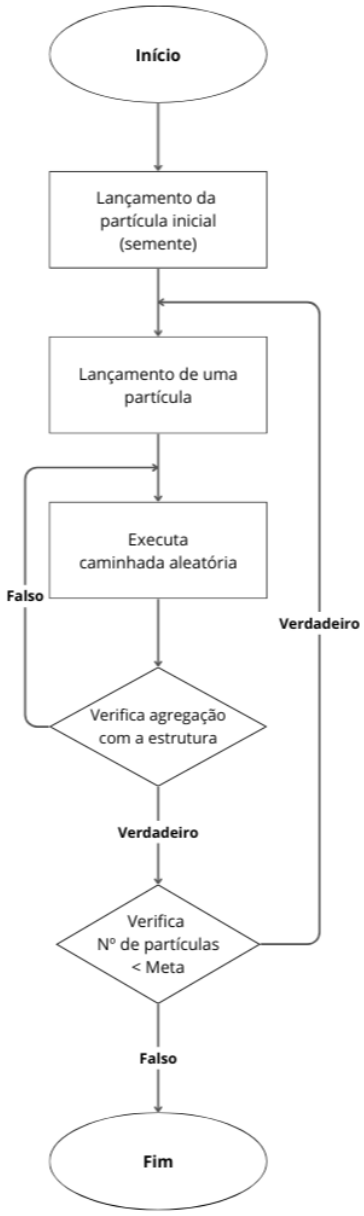
4.1 AGREGAÇÃO LIMITADA POR DIFUSÃO

O DLA é um método proposto originalmente por Witten e Sander (1981) para descrever o processo de formação de agregados aleatórios. Esse modelo é um fractal randômico frequentemente utilizado para a simulação de fenômenos da natureza, como por exemplo, a formação de flocos de neve, crescimento de colônias bacterianas, crescimento de recifes de corais, entre outros (BRAGA; RIBEIRO, 2011).

A simulação computacional do DLA é realizada utilizando uma grade bidimensional ou tridimensional. O processo inicia-se com a fixação de uma única partícula no centro da grade, que é utilizada como o núcleo da estrutura. Após isso, novas partículas são geradas em posições aleatórias (BRAGA; RIBEIRO, 2011). Cada uma dessas partículas executa uma caminhada aleatória na rede, onde, a cada passo de tempo, a partícula se desloca de sua posição atual para um dos sítios vizinhos, escolhido com igual probabilidade. Este tipo de movimento é uma aproximação discreta do movimento browniano (ZSAKI, 2016).

A caminhada aleatória prossegue até que a partícula encontre um local adjacente a uma partícula que já faz parte da estrutura. Nesse momento, ocorre a agregação, onde o movimento é finalizado e a partícula é fixada permanentemente na estrutura. Esta etapa de agregação é irreversível, ou seja, uma vez anexada, a partícula não pode ser removida ou reposicionada (WITTEN; SANDER, 1981). Este ciclo continua até que o aglomerado atinja um tamanho pré-determinado ou um número específico de partículas seja incorporado. O fluxograma do algoritmo é apresentado na Figura 16.

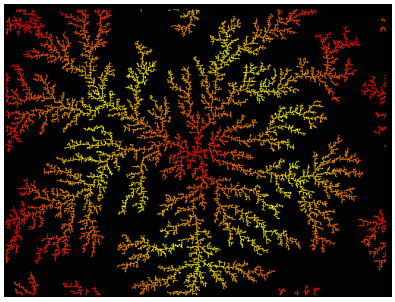
Figura 16 – Algoritmo DLA



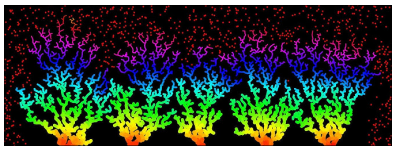
Fonte: Adaptado de Braga e Ribeiro (2011).

A forma específica de um agregado DLA é extremamente sensível a diversos parâmetros da simulação, como a estrutura inicial e as regras de agregação. As condições iniciais, como o número, a forma e a disposição espacial das partículas que iniciam o processo de agregação, são fundamentais. Uma única partícula central, geralmente, leva a um crescimento radial (WITTEN; SANDER, 1981), enquanto múltiplas sementes podem resultar em estruturas policêntricas, onde diferentes agregados crescem e podem eventualmente interagir ou competir pelo espaço (ZSAKI, 2016). A Figura 17a mostra uma estrutura gerada com uma única partícula inicial no centro da grade e a Figura 17b mostra uma estrutura com cinco partículas inicialmente colocadas na parte superior da grade.

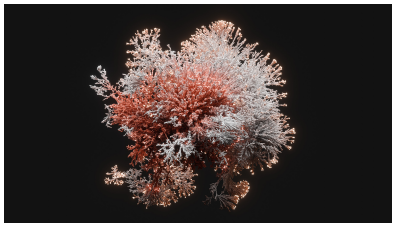
Figura 17 – Exemplos de fractais gerados a partir do DLA.



(a) Fractal 2D com uma partícula inicial central.



(b) Fractal 2D com cinco partículas iniciais.



(c) Fractal 3D visualizado no software *Blender*.

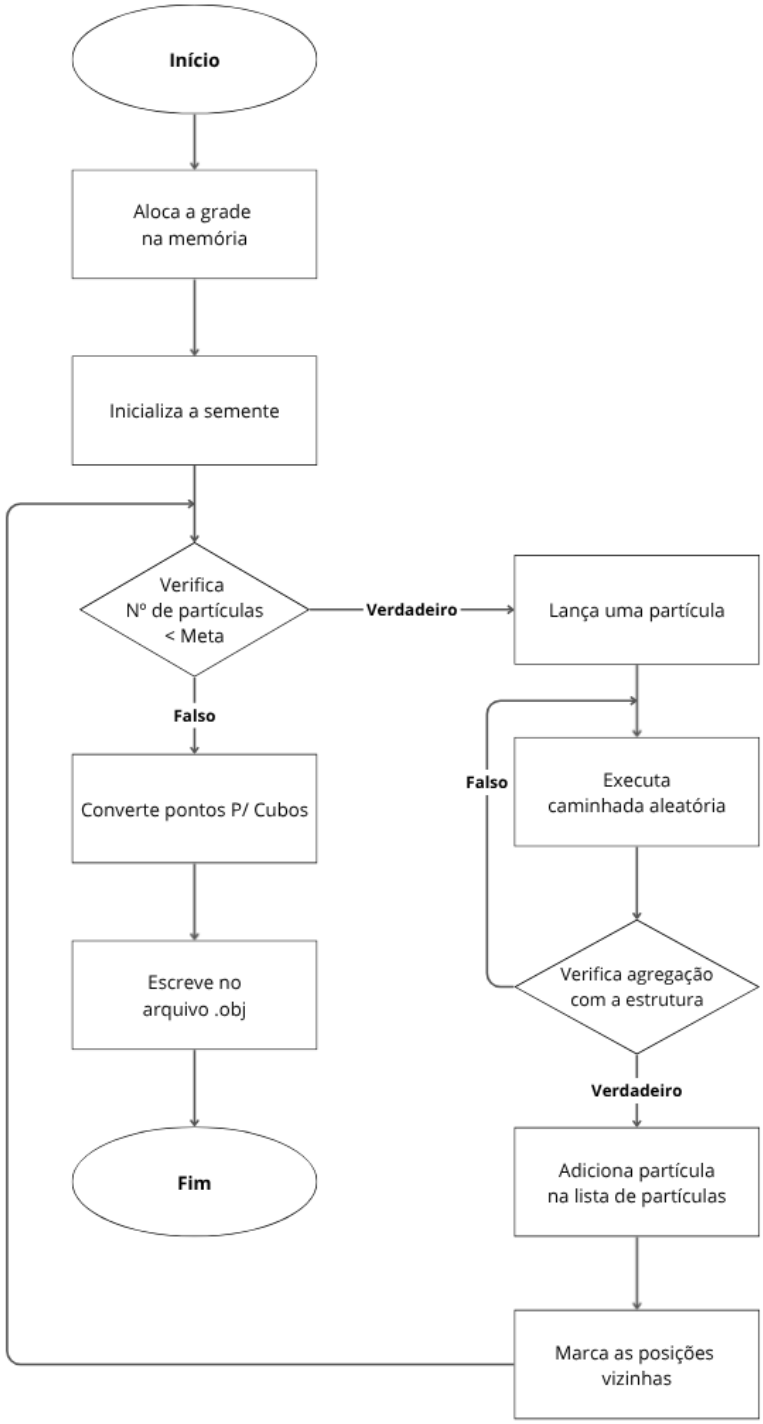
Fonte: Chr. Clemens Lee (2006), Programming Chaos (2023) e Samuel (sml) (2025), respectivamente.

As regras de agregação, como a probabilidade de uma partícula se fixar ao agregado após o contato, influenciam na densidade e a textura do agregado. Uma alta taxa de agregação, onde a partícula gruda no primeiro contato, favorece estruturas mais abertas e ramificadas (Figura 17a), pois as pontas mais externas são mais propensas a capturar novas partículas. Por outro lado, uma baixa taxa de agregação permite que as partículas realizem múltiplos contatos antes de se fixarem, o que possibilita com que as partículas alcancem pontos mais profundos da estrutura, o que resulta em agregados mais compactos e densos (Figura 17c) (WITTEN; SANDER, 1981).

4.2 IMPLEMENTAÇÃO SEQUENCIAL

A implementação sequencial foi desenvolvida na linguagem de programação C++. A Figura 18 apresenta um fluxograma do algoritmo sequencial. O mesmo inicia com a alocação de uma grade tridimensional, que representa o espaço de simulação, e com a inserção da semente em seu centro. Na etapa seguinte, o algoritmo executa um laço principal que continua até que o número desejado de partículas seja agregado. As coordenadas de cada partícula agregada são salvas em um vetor unidimensional. A cada iteração, uma nova partícula é criada em uma posição aleatória no espaço de simulação e inicia uma "caminhada aleatória". O movimento da partícula é realizado até que uma das três condições seja atendida: ela encontra uma posição adjacente à estrutura já formada e se agrega a ela; exceda um número máximo de passos sendo descartada; ou atinja uma posição fora dos limites da grade. Uma vez que uma partícula se agrega, ela é adicionada ao vetor de partículas agregadas e as posições vizinhas a ela são marcadas como locais potenciais para futuras agregações na grade tridimensional.

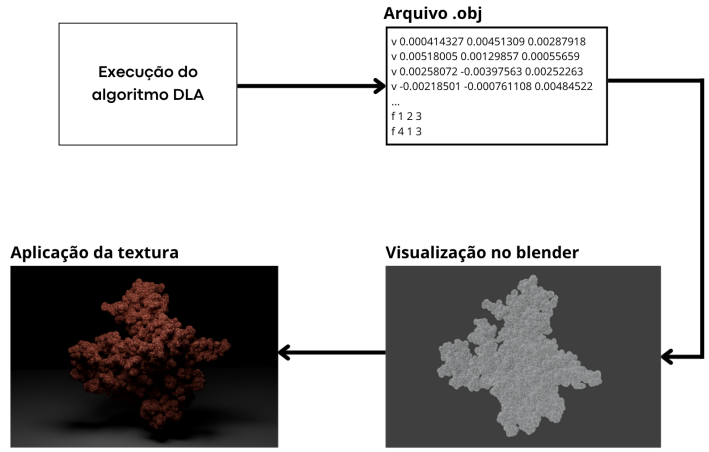
Figura 18 – Fluxograma do algoritmo sequencial



Fonte: O Autor (2025).

Após atingir o número de partículas esperado, o algoritmo produz um arquivo de saída no formato *Wavefront* (.OBJ), o qual pode ser posteriormente visualizado no software *Blender*. Por fim, aplica-se uma textura ao objeto no software *Blender* para finalizar a geração do coral. A Figura 19 apresenta o processo de de visualização do fractal.

Figura 19 – Fluxograma da geração de Corais



Fonte: O Autor (2025).

4.2.1 Grade Tridimensional de Simulação

A simulação é realizada em uma grade tridimensional, que representa o espaço onde o fractal é gerado. Para otimizar e facilitar o gerenciamento de memória, a grade de dimensões $N \times N \times N$ foi implementada como um vetor unidimensional de N^3 elementos, onde N corresponde à dimensão da grade. O mapeamento de uma coordenada tridimensional (x, y, z) para um índice no vetor unidimensional é realizado pela Equação 4.1.

$$index = z \times N^2 + y \times N + x \tag{4.1}$$

Cada posição da grade armazena um número inteiro contendo o estado da posição. O valor *0* representa uma posição livre. O valor *1* indica uma posição adjacente a uma partícula já agregada à estrutura, ou seja, é uma posição possível para uma partícula se agregar. E, por fim, o valor *2* indica uma posição ocupada por uma partícula que já pertence à estrutura do fractal. Uma representação bidimensional da estrutura da grade pode ser visualizada na Figura 20.

Figura 20 – Representação 2D da estrutura da grade

0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0
0	0	0	1	1	2	1	0	0
0	0	1	2	2	2	1	0	0
0	1	2	2	2	2	1	0	0
0	0	1	1	2	1	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0

Fonte: O Autor (2025).

4.2.2 Movimento de uma Partícula

O movimento de cada partícula é realizado através da função `randomWalk`, apresentada no Algoritmo 6. Essa função realiza o cálculo do movimento aleatório da partícula e verifica a possibilidade de agregação. Para cada passo da caminhada, a partícula é deslocada em uma direção aleatória do espaço tridimensional da grade (linha 6 a linha 12). Após o deslocamento, verifica-se se a posição atual da partícula corresponde a um ponto de agregação na estrutura (linha 24). Em caso, positivo a função retorna à posição atual. Em caso contrário, a partícula continuara a movimentação até: agregar-se à estrutura; chegar ao limite de passos; ou sair da área da grade.

Algoritmo 6 – Algoritmo da caminhada aleatória sequencial

```
1  Particula randomWalk(int limite_passos) {
2      Particula pos_atual;
3      for (int passo = 0; passo < limite_passos; passo++) {
4
5          // Move aleatoriamente em uma das 6 direcoes
6          int dir = rand() % 6;
7          if (dir == 0) pos_atual.x++;
8          else if (dir == 1) pos_atual.x--;
9          else if (dir == 2) pos_atual.y++;
10         else if (dir == 3) pos_atual.y--;
11         else if (dir == 4) pos_atual.z++;
12         else if (dir == 5) pos_atual.z--;
13
14         // Verifica limites
15         if (pos_atual.x < 0 || pos_atual.x >= TAMANHO_GRADE ||
16             pos_atual.y < 0 || pos_atual.y >= TAMANHO_GRADE ||
17             pos_atual.z < 0 || pos_atual.z >= TAMANHO_GRADE) {
18             return Particula(-1, -1, -1);  // Saiu da grade
19         }
20
21         int idx = calcIndex(pos_atual.x, pos_atual.y, pos_atual.z);
22
23         // Verifica agregacao
24         if (grade[idx] == 1) {
25         // Tenta agregar de acordo com a probabilidade de agregacao
26             if (tentaAgregacao()) {
27                 return pos_atual;
28             }
29         }
30     }
31     return Particula(-1, -1, -1);  // Finaliza particula sem agregacao
32 }
```

Fonte: O Autor (2025).

Ao final da simulação, tem-se um vetor contendo todas as partículas agregadas à estrutura do fractal, onde cada elemento do vetor armazena as coordenadas de uma partícula. Esse vetor é então utilizado para gerar um arquivo no formato *Wavefront OBJ*, para uma posterior visualização.

4.2.3 Geração do Arquivo de Saída

O arquivo de saída utiliza o formato *Wavefront OBJ*. Este formato de arquivo foi desenvolvido na década de 90 pela empreas *Wavefront Technologies* e foi utilizado no software *Advanced Visualizer* (LOC, 2025). O formato *OBJ* estabeleceu-se como um padrão na indústria de computação gráfica devido a sua simplicidade e estrutura aberta (BLENDER, 2025b).

Um exemplo de um arquivo no formato *Wavefront OBJ* é ilustrado no Algoritmo 7. O arquivo é formado por duas listas, uma de vértices e de faces. Um vértice é representado pela sintaxe `v x y z`, onde o caractere `v` indica que a linha define um vértice, e as variáveis `x`, `y` e `z` correspondem às coordenadas em ponto flutuante do vértice (LOC, 2025). As faces são usadas para construir os polígonos que formam a superfície do modelo. Uma face é definida pela sintaxe `f a b c`, onde o caractere `f` indica que a linha representa uma face, e os valores `a`, `b` e `c` são os índices dos vértices que formam a face (BLENDER, 2025b).

Algoritmo 7 – Arquivo no formato *Wavefront OBJ*

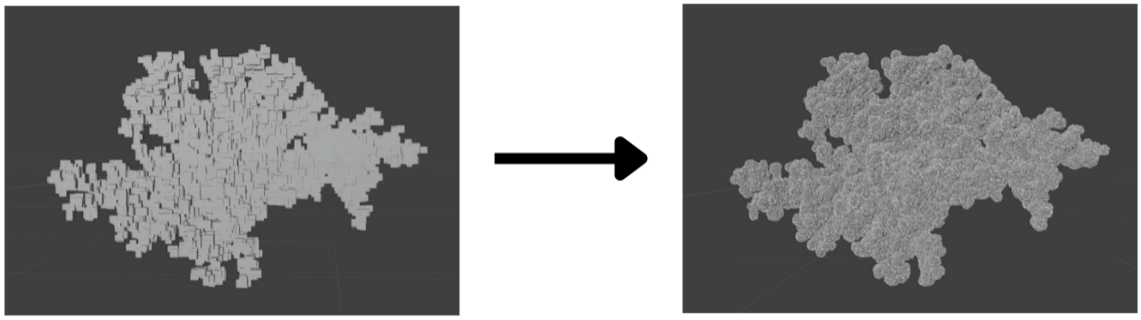
```
1
2 v 0.000414327 0.00451309 0.00287918
3 v 0.00518005 0.00129857 0.00055659
4 v 0.00258072 -0.00397563 0.00252263
5 v -0.00218501 -0.000761108 0.00484522
6 ...
7 f 1 2 3
8 f 4 1 3
```

Fonte: O Autor (2025).

Para a conversão do vetor de partículas em uma figura tridimensional, cada partícula foi representada como um cubo no arquivo *OBJ*. Dessa forma, as coordenadas existentes no vetor foram utilizadas como posições centrais desses cubos. A partir de cada centro, e considerando uma aresta de tamanho aleatório, são calculados os oito vértices que definem o cubo. Em seguida, são geradas as doze faces triangulares que conectam esses vértices, resultando na superfície fechada do cubo. O resultado desse procedimento é uma estrutura que representa o agregado.

No entanto, a imagem gerada apresenta uma aparência quadriculada (Figura 21a). Assim, foi implementado um método de suavização, cujo objetivo é gerar volumes mais arredondados, conferindo ao fractal uma aparência mais natural. Neste caso, diversos cubos foram gerados na mesma posição, cuja sobreposição produz um volume visualmente mais denso. Além disso, aplica-se uma rotação aleatória nos três eixos x, y e z, fazendo com que as arestas e faces dos cubos percam o alinhamento e gerem uma forma mais arredondada (Figura 21b).

Figura 21 – Coral Suavizado



Fonte: O Autor (2025).

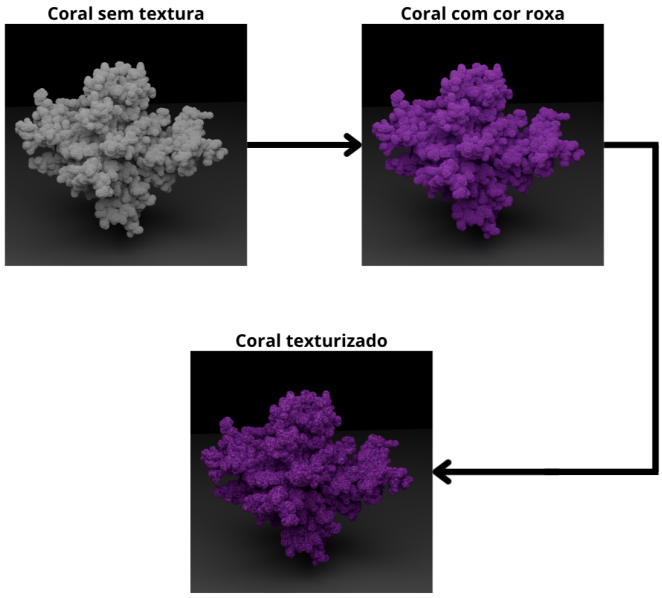
4.2.4 Visualização do Fractal

O arquivo de saída no formato *OBJ* descreve a estrutura tridimensional do coral. Porém, o arquivo consiste em um modelo monocromático e pouco realista. Para transformar essa estrutura em uma representação visual realista de um coral, foi utilizada uma etapa de pós-processamento no software de modelagem 3D *Blender*. A texturização do coral foi realizada por meio de um sistema de *shaders* procedurais no *Blender*[1](ALDA, 2023).

Inicialmente, aplicou-se uma cor base e, em seguida, adicionou-se uma textura procedural do tipo *Noise Texture*, que introduz pequenas irregularidades na superfície, criando relevo e tornando o coral visualmente mais realista (BLENDER, 2025a). A iluminação da cena foi ajustada para realçar o volume e as variações resultantes da textura. A Figura 22 apresenta o coral em cada etapa da texturização: sem textura (sua forma nativa), com a aplicação da cor base e, por fim, com a textura gerada pelo *Noise Texture*.

[1] Shaders são programas responsáveis por determinar como a superfície de um objeto reage à luz, controlando características como cor, brilho e relevo

Figura 22 – Etapas de texturização do Coral



Fonte: O Autor (2025).

4.3 IMPLEMENTAÇÃO PARALELA

Para a paralelização do processo de geração do fractal, foi utilizada a plataforma de computação paralela CUDA, na versão 12.8. Para o desenvolvimento da implementação paralela foram identificados os trechos da implementação com maior custo computacional por meio do perfilamento do código-fonte, utilizando a ferramenta *gprof* (FENLASON; STALLMAN, 1998). O Quadro 3 resume os resultados para uma simulação com 10.000 partículas. Observa-se que a função `randomWalk`, responsável pela simulação do movimento e agregação das partículas, representa a tarefa com maior custo computacional, consumindo 88.22% do tempo total de execução. Por outro lado, a etapa de pós-processamento e geração do arquivo de saída realizada pela função `pointsToCubes` corresponde a uma pequena fração do tempo, ocupando apenas 0.11% do tempo total de execução. Desta forma, não é necessária a paralelização dessa função.

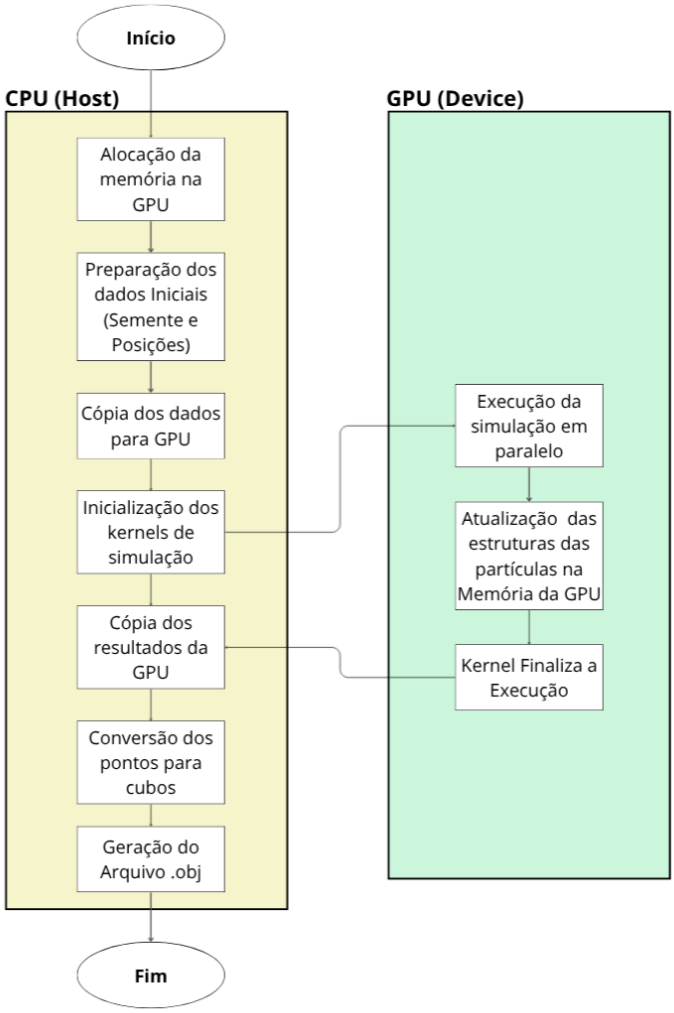
Quadro 3 – Análise de tempo de execução das funções na versão sequencial

Função	**Tempo (s)**	**% do Tempo Total**	**Nº de Chamadas**
`randomWalk()`	41.02	88.22%	305.465
`getIndex()`	1.88	4.04%	∼3.89 bilhões
`std::vector::operator[]`	1.33	2.87%	∼3.89 bilhões
`pointsToCubes()`	0.05	0.11%	1
Outras Funções	∼2.22	∼4.76%	-
Total	∼46.50	100%	-

Fonte: O Autor (2025).

A implementação paralela foi desenvolvida conforme descrito em Zsaki (2016). Nele, cada partícula é mapeada para uma *thread*, possibilitando que milhares de partículas realizem suas caminhadas aleatórias de forma concorrente. Assim, o laço sequencial que processava uma partícula por vez foi substituído por um único *kernel* que executa a simulação para todas as partículas simultaneamente. Na Figura 23, tem-se um fluxograma da implementação paralela. No lado esquerdo, em amarelo, estão as tarefas que são executadas pela CPU. À direita, em verde, estão as tarefas executadas pela GPU.

Figura 23 – Fluxograma do programa CUDA



Fonte: O Autor (2025).

O Algoritmo 8 apresenta o código referente à alocação das estruturas na memória da GPU. O Inicialmente é utilizada a função `cudaMalloc` para a alocação da grade tridimensional e da lista de partículas na memória da GPU (linhas 2 a 5). Após isso, as estruturas de dados e a semente são inicializadas na CPU e copiadas para a GPU. A cópia é realizada através da função `cudaMemcpy` em conjunto com a diretiva `cudaMemcpyHostToDevice`, que especifica que a transferência é feita da memória do *host* para a memória do *device* (linhas 8 e 11).

Por fim, é realizada a execução do *kernel* de simulação (`randomWalk`) (linha 15).

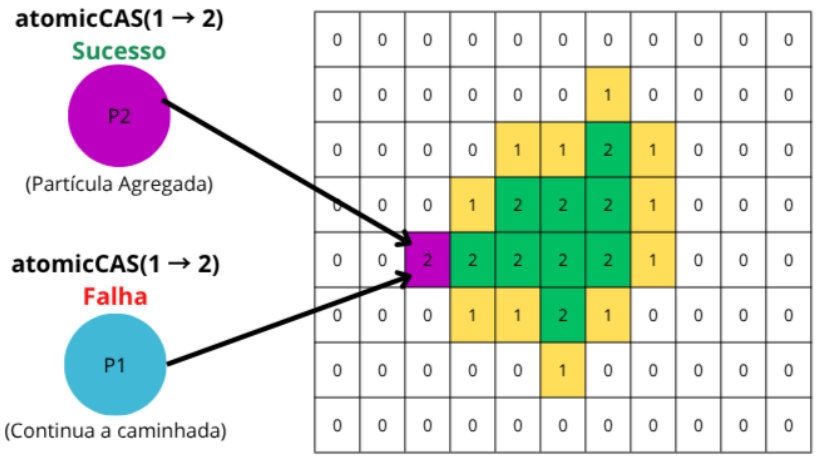
Algoritmo 8 – Alocação das estruturas na GPU e chamada do *kernel*

```
1  // Alocacao da memoria na GPU
2  cudaMalloc(&d_particles, NUM_PARTICLES * sizeof(Point));
3  cudaMalloc(&d_grid, grid_size);
4  cudaMalloc(&d_frozen_points, NUM_PARTICLES * sizeof(Point));
5  cudaMalloc(&d_frozen_count, sizeof(int));
6
7  // Copia dos dados da grade e particulas da CPU para a GPU
8  cudaMemcpy(d_grid + nidx, &val_dock,
9             sizeof(int), cudaMemcpyHostToDevice);
10
11 cudaMemcpy(d_particles, h_particles.data(),
12             NUM_PARTICLES * sizeof(Point), cudaMemcpyHostToDevice);
13
14 // Chamada do Kernel randomWalk
15 randomWalk<<<blocks, threads_per_block>>>(...);
```

Fonte: O Autor (2025).

Na GPU, cada uma das *threads* inicia o processo de caminhada aleatória. Neste caso, múltiplas *threads* podem tentar modificar simultaneamente uma mesma posição da grade tridimensional, o que acarretaria em resultados inconsistentes. Para resolver esse problema, foram utilizadas operações atômicas. Quando uma *thread* identifica um local válido para agregação (valor 1), é realizada uma operação atômica para a alteração do estado dessa posição para ocupado (valor 2). As demais *threads* que disputavam o mesmo local de memória falharão e continuarão sua caminhada aleatória.

Figura 24 – Ilustração da operação atômica



Fonte: O Autor (2025).

O Algoritmo 9 apresenta o código referente à implementação da operação atômica. A função `atomicCAS` foi aplicada no momento da agregação: quando uma *thread* identifica um local de agregação, é executada uma chamada a `atomicCAS` (linha 6) para alterar a posição para ocupado. As demais *threads* não conseguem acessar essa posição e continuam a sua execução. Além disso, a operação `atomicAdd` (linha 12) foi utilizada para incrementar o contador global de partículas agregadas.

Algoritmo 9 – Operações atômicas dentro do *kernel* da caminhada aleatória

```
1  // Particula possivelmente congelada
2  if (current_val == 1) {
3      if (prob_atual <= prob_de_agregacao) {
4
5          // Tenta congelar a particula
6          int old = atomicCAS(&grid[idx], 1, 2);
7
8          // Se conseguiu congelar
9          if (old == 1) {
10
11             // Adiciona a particula ao array de congeladas
12             int slot = atomicAdd(frozen_count, 1);
13
14             for (int i = 0; i < 6; i++) {
15                 ...
16
17                 // Marca os vizinhos como
18                 // possiveis pontos de agregacao
19                 atomicCAS(&grid[nidx], 0, 1);
20             }
21         }
22         ...
23     }
24     // Se nao conseguiu congelar, continua ativo
25 }
```

Fonte: O Autor (2025).

Ao final da execução do *kernel*, uma chamada ao comando `cudaMemcpy`, em conjunto com a diretiva `cudaMemcpyDeviceToHost`, é executada para transferir a lista de partículas agregadas da memória da GPU para a memória da CPU, conforme apresentado no Algoritmo 10. Por fim, a CPU executa a geração do arquivo `.OBJ`, concluindo o processo de geração do fractal.

Algoritmo 10 – Cópia da memória da GPU para a CPU

```
1 // Copia numero total de particulas agregadas da GPU para a CPU
2 int final_count;
3 cudaMemcpy(&final_count, d_frozen_count,
4             sizeof(int), cudaMemcpyDeviceToHost);
5
6 // Copia a lista particulas agregadas da GPU para a CPU
7 vector<Point> h_frozen_from_gpu(final_count);
8 cudaMemcpy(h_frozen_from_gpu.data(), d_frozen_points,
9             final_count * sizeof(Point), cudaMemcpyDeviceToHost);
```

Fonte: O Autor (2025).

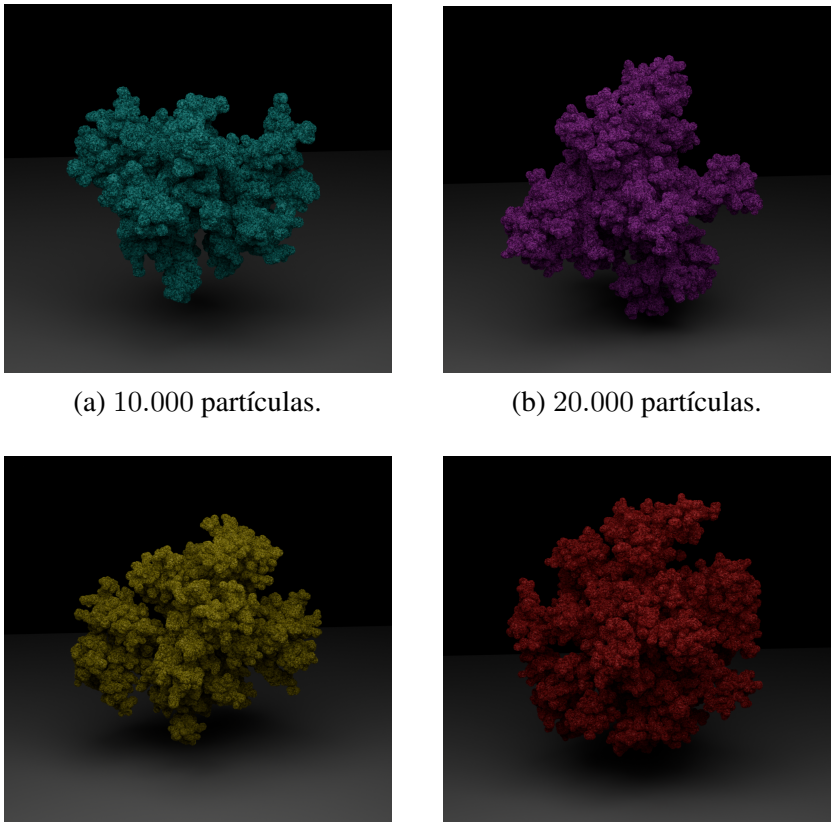
5 TESTES E RESULTADOS OBTIDOS

A geração dos fractais de corais é sensível à estrutura inicial, às regras de agregação e às condições iniciais das partículas. Neste capítulo, são analisadas as influências desses parâmetros sobre os fractais gerados. Nos testes, todas as simulações do algoritmo DLA foram executadas em uma grade tridimensional com dimensões de $350 \times 350 \times 350$. Em seguida, apresentados os resultados obtidos com a implementação paralela em GPU.

5.1 INFLUÊNCIA DO NÚMERO DE PARTÍCULAS

Na Figura 25 é possível visualizar os fractais gerados utilizando 10.000, 20.000, 30.000 e 40.000. Para geração dos fractais foram utilizadas 50 unidades de distância da semente para o nascimento das partículas e uma probabilidade de agregação de 100%. Como pode ser observado, os fractais com um número maior de partículas são mais densos. De fato, com mais partículas disponíveis para agregação, o fractal preenche um volume significativamente maior da grade e as ramificações tornam-se mais espessas.

Figura 25 – Número de partículas



(a) 10.000 partículas. (b) 20.000 partículas.

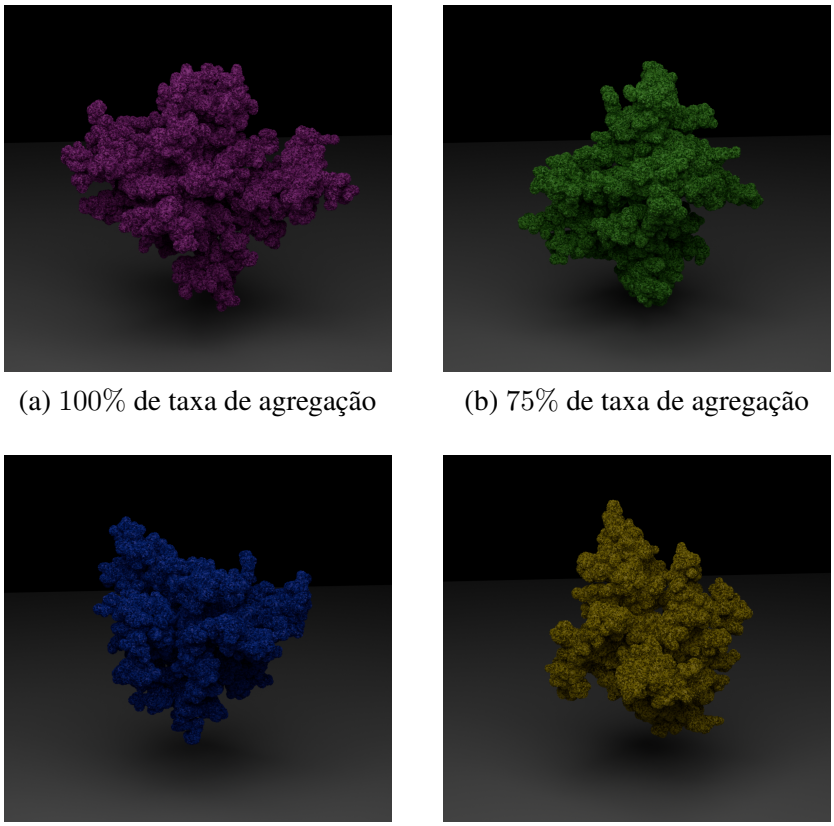
(c) 30.000 partículas. (d) 40.000 partículas.

Fonte: O Autor (2025).

5.2 PROBABILIDADE DE AGREGAÇÃO DAS PARTÍCULAS

A Figura 26 apresenta o processo de geração do fractal utilizando taxas de agregação de 25%, 50%, 75% e 100%. Observa-se que a redução dessa taxa resulta em corais mais compactos, com menos extensões delicadas e um volume mais concentrado próximo ao núcleo. Isso ocorre porque as partículas se originam mais próximas do centro da estrutura, o que dificulta a formação de novos ramos.

Figura 26 – Taxa de Agregação



(a) 100% de taxa de agregação (b) 75% de taxa de agregação

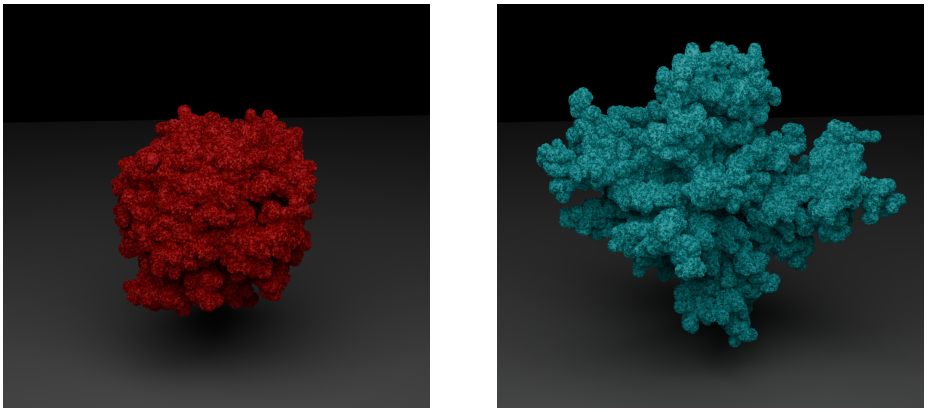
(c) 50% de taxa de agregação (d) 25% de taxa de agregação

Fonte: O Autor (2025).

5.3 DISTÂNCIA DE NASCIMENTO DAS PARTÍCULAS EM RELAÇÃO À SEMENTE

A Figura 27 apresenta a diferença dos fractais utilizando a distância de nascimento de 20 e 50 unidades de distância da semente inicial. Observa-se que uma curta distância causa um agrupamento mais rápido, o que acarreta um coral menor e mais denso. Por outro lado, o aumento dessa distância força as partículas a iniciarem a caminhada aleatória mais longe da estrutura central. Assim, quanto maior a distância de nascimento das partículas em relação à semente mais esparsa é a estrutura, exibindo ramificações mais longas.

Figura 27 – Distância de nascimento



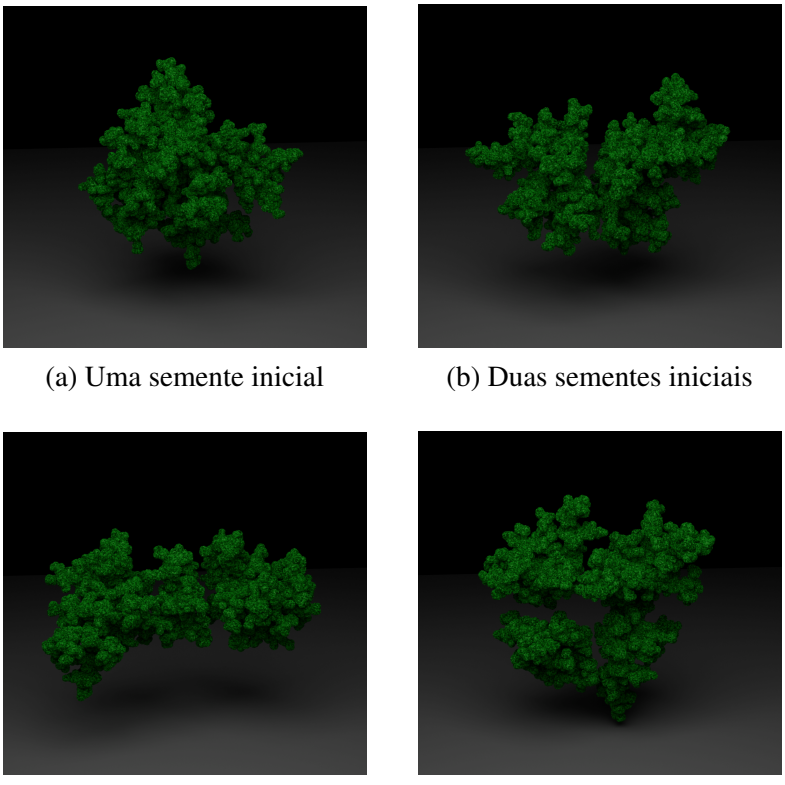
(a) Distância de 20 unidades. (b) Distância de 50 unidades.

Fonte: O Autor (2025).

5.4 MÚLTIPLAS SEMENTES INICIAIS

A Figura 28 apresenta a geração de corais com 1 até 4 partículas como sementes. Neste caso, cada semente atua como um núcleo de agregação independente, gerando sua própria estrutura fractal. Observa-se que a estrutura final exibe múltiplos agregados que cresceram, interagiram e começaram a se fundir, criando uma colônia de corais composta.

Figura 28 – Número de sementes



(a) Uma semente inicial (b) Duas sementes iniciais

(c) Três sementes iniciais (d) Quatro sementes iniciais.

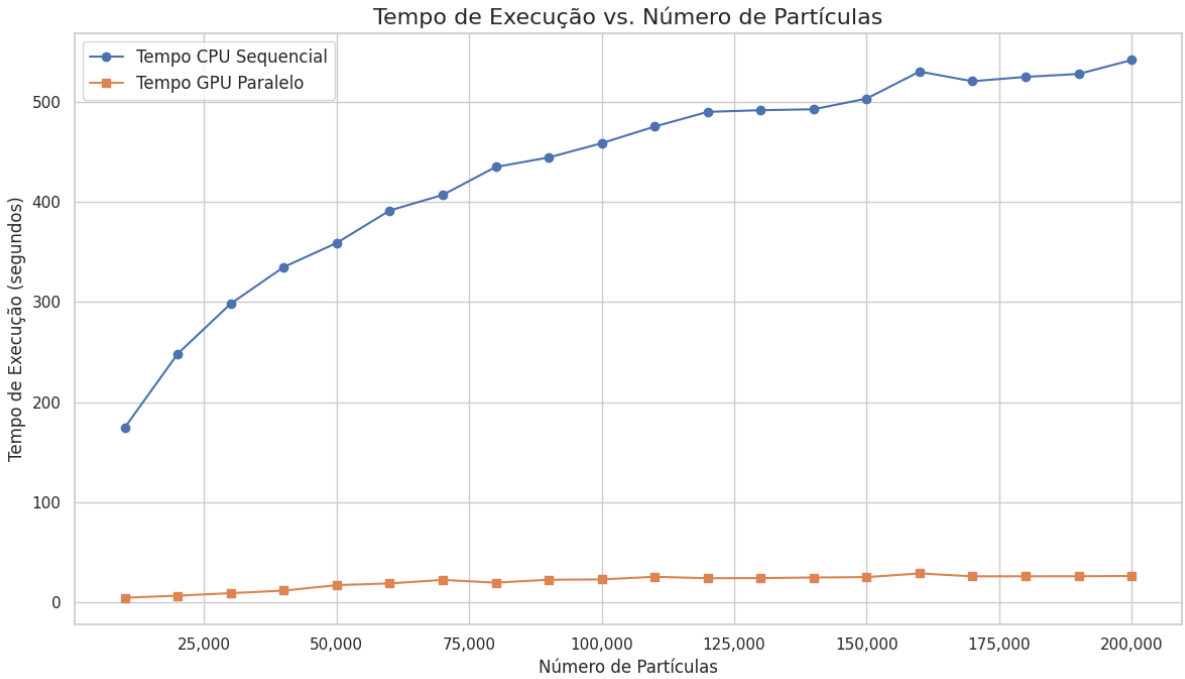
Fonte: O Autor (2025).

5.5 AVALIAÇÃO DA IMPLEMENTAÇÃO PARALELA

Os testes foram realizados em um computador com um processador *Intel Core i5-14600K*, com 14 núcleos de processamento, operando a uma frequência de $3,5$ GHz. O processador possui uma *cache* L1 de 80 KB, *cache* L2 de 2 MB e *cache* L3 compartilhada de 24 MB. Além disso, o computador dispõe de 32 GB de memória RAM DDR5, operando a 4800 MHz. O armazenamento é feito em um SSD NVMe de 1 TB. O sistema operacional utilizado foi o Ubuntu 24.04.2 LTS, virtualizado em um subsistema WSL no sistema operacional Windows 11 Pro de 64 bits. O computador possui ainda uma GPU NVIDIA RTX 4060 Ti, baseada na arquitetura *Ada Lovelace*. A placa possui 4.352 núcleos CUDA, distribuídos em 34 SMs. Além disso, conta com uma memória global de 8 GB operando a 2.250 MHz, interface de 128 *bits* e largura de banda de 288 GB/s. A placa ainda dispõe de uma *cache* L1 de 128 KB e uma *cache* L2 de 32 MB.

Para os testes de foram gerados fractais utilizando: uma grade $700 \times 700 \times 700$; uma partícula centralizada na grade como semente inicial; 90 unidades de distância da semente para o nascimento das partículas; e uma probabilidade de agregação de 100%. Na Figura 29, apresenta-se um comparativo entre os tempos de execução sequencial e paralela em GPU do algoritmo DLA, utilizando de 10.000 a 200.000 partículas, com incremento de 10.000 partículas por teste. Para cada conjunto de partículas foram realizadas 5 execuções, considerando-se como tempo final a média aritmética dos resultados obtidos.

Figura 29 – Tempos de execução

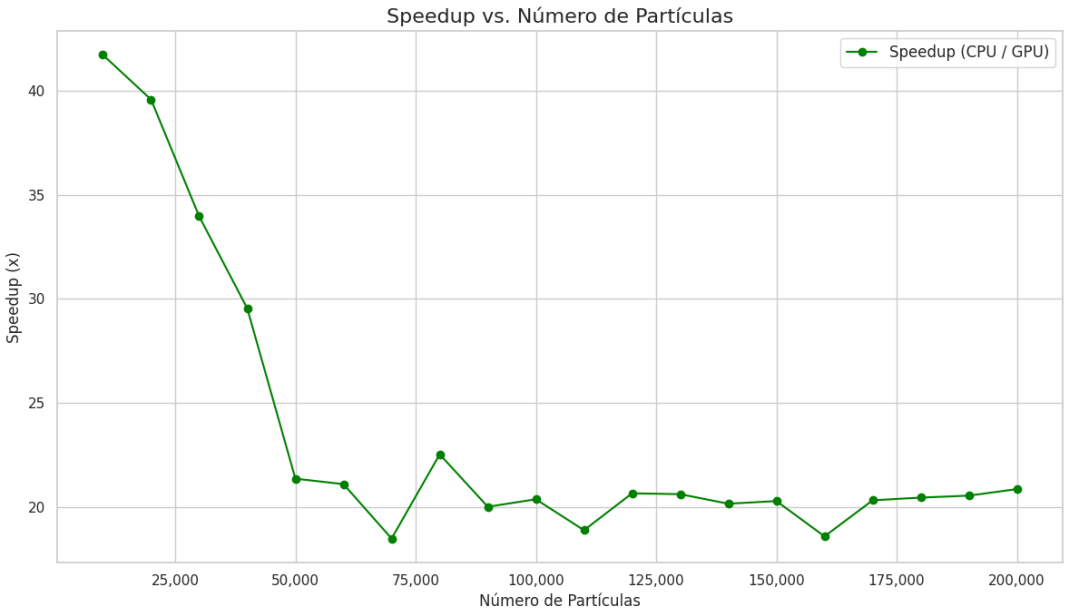


Fonte: O Autor (2025).

Considerando a versão sequencial do algoritmo, observa-se na Figura 29 que o tempo de execução da implementação sequencial apresenta um crescimento considerável nos testes iniciais. À medida que o número de partículas aumenta, esse crescimento tende a se reduzir. Esse comportamento está relacionado à própria natureza do algoritmo DLA: quanto maior o número de partículas agregadas, maior se torna a estrutura e, consequentemente, a região de possível agregação das novas partículas. Dessa forma, o tempo médio de caminhada aleatória diminui, reduzindo o tempo necessário para que cada partícula se agregue ao coral.

Observa-se ainda, na Figura 29, que a implementação paralela em GPU apresenta tempos de execução significativamente menores que os obtidos na versão sequencial. A Figura 30 apresenta o resultado do cálculo do *speedup*[1] entre a CPU e a GPU. É possível observar que o *speedup* e diminui até estabilizar em torno de 20. A redução observada no *speedup* deve-se ao aumento da concorrência entre as *threads*. De fato, com o crescimento do número de partículas, ocorre um acréscimo no número de *threads* ativas e de acessos à memória, o que eleva o custo das operações atômicas e resulta em perda de desempenho. Além disso, observa-se uma estabilização do *speedup* a partir das 50.000 partículas. Essa estabilidade está relacionada à arquitetura da placa de vídeo utilizada. A RTX 4060 Ti possui um máximo de 52.224 *threads* que podem ser executadas simultaneamente (NVIDIA, 2025b). Assim, acima desse limite, não há ganho adicional de processamento, pois as *threads* excedentes permanecem em espera até serem executadas.

Figura 30 – *Speedup* da implementação paralela em GPU



Fonte: O Autor (2025).

[1] O *speedup* é a métrica que quantifica o ganho de desempenho da execução em GPU em relação à versão sequencial do programa, sendo calculado como $Speedup = \frac{T_{seq}}{T_{gpu}}$.

6 CONSIDERAÇÕES FINAIS

Neste trabalho foi desenvolvida uma implementação paralela em GPU para a geração de fractais de corais. Optou-se pela utilização da arquitetura de GPU, devido ao seu elevado poder computacional em comparação à CPU, resultante do grande número de núcleos de processamento disponíveis. Além disso, essa arquitetura é amplamente disponível em computadores pessoais. O aumento da demanda por jogos de PC contribuiu para a popularização e redução do custo das GPUs, que hoje estão presentes em um grande número de computadores pessoais.

Para a implementação, foi utilizada a linguagem de programação C++ e a plataforma CUDA da NVIDIA. Para o desenvolvimento em GPUs, uma alternativa era utilizar a plataforma *OpenCL*, porém optou-se pela utilização da plataforma CUDA, pois essa foi desenvolvida especificamente para as GPUs da NVIDIA. Além disso, CUDA apresenta uma sintaxe mais amigável e intuitiva, o que facilita a escrita e compreensão do código.

Para a geração dos fractais de corais, optou-se pelo método DLA, por sua ampla utilização em aplicações que visam simular fenômenos de crescimento natural, como por exemplo, a formação de flocos de neve e crescimento de colônias bacterianas. Além disso, esse método pode ser eficientemente paralelizado, pois o movimento de múltiplas partículas pode ser calculado de forma independente.

Os testes realizados permitiram verificar visualmente que a geração dos fractais de corais é sensível a diferentes parâmetros configuráveis. O aumento do número de partículas resulta em corais mais densos e maiores, pois elas preenchem um volume maior da grade. As taxas de agregação influenciam diretamente a formação das ramificações, uma vez que taxas menores permitem que as partículas penetrem mais profundamente na estrutura. A distância de nascimento das partículas afeta o tamanho das ramificações, já que sementes mais próximas da estrutura se agregam mais rapidamente. Por fim, a adição de novas sementes iniciais gera núcleos de agregação independentes, que podem crescer separadamente e, posteriormente, fundir-se com o restante da estrutura.

Uma limitação do trabalho desenvolvido consiste na visualização dos fractais de corais, atualmente realizada por meio de softwares de terceiros, como o *Blender*. Essa limitação poderia ser abordada com a utilização de bibliotecas de programação gráfica, como o *OpenGL*, possibilitando a texturização e visualização das estruturas dos corais diretamente em um software próprio.

Para os testes de desempenho foram comparados os tempos de execução entre a implementação sequencial e a implementação paralela em GPU. Os resultados obtidos demonstram que a versão paralela em GPU apresentou um tempo de execução menor em relação à implementação sequencial na CPU. No entanto, observou-se uma redução no valor do *speedup*, seguida de uma estabilização. Esse comportamento ocorre devido ao limite máximo de *threads* que podem estar ativas simultaneamente na GPU utilizada para testes. Assim, ao se ultrapassar esse limite, a GPU não é capaz de executar todas as *threads* de forma simultânea, fazendo com que as demais aguardem até que núcloes de processamento sejam liberados.

6.1 TRABALHOS FUTUROS

Como trabalhos futuros sugere-se:

- Paralelização dos algoritmos *Reaction–diffusion system* e *L-systems* utilizando a plataforma CUDA.
- Visualização e texturização das estruturas utilizando ferramentas de computação gráfica como *OpenGL*.
- Execução do algoritmo paralelo em GPUs da NVIDIA que permitem um número maior de *threads* concorrentes.
- Avaliação do consumo energético da implementação paralela em GPU, a fim de comparar sua eficiência com a execução em CPU.

REFERÊNCIAS

ABD-EL-BARR, M.; EL-REWINI, H. **Fundamentals of Computer Organization and Architecture**. Wiley, 2005. (Wiley Series on Parallel and Distributed Computing). ISBN 9780471478331. Disponível em: <https://books.google.com.br/books?id=Wh3k6Oc_Tw8C>.

ALDA, A. Introduction to shaders. In: ____. [S.l.]: Apress, 2023. p. 1–21. ISBN 978-1-4842-9671-4.

ALMEIDA, E.; SANTOS, T. Uma breve introdução ao conjunto de cantor. **Revista de Matemática da Universidade Federal de Ouro Preto**, v. 1, p. 60–65, 12 2017.

ARSIE, K. C. DimensÃo espacial. **PET-Matemática-UFPR**, 2008. Disponível em: <https://docs.ufpr.br/~ewkaras/ic/karla08.pdf>.

ASSIS, T. A. de. **Geometria fractal: propriedades e características de fractais ideais**. Revista Brasileira de Ensino de Física, 2008. Disponível em: <https://www.scielo.br/j/rbef/a/NkxTkgKJJdBX6Zy95zWHZkG/?format=pdf&lang=pt>.

AYIRLI, M. B. Determining different plant leaves' fractal dimensions: a new approach to taxonomical study. **Bangladesh J. Bot.**, v. 43, p. 267–275, 2014. Disponível em: <https://dspace.balikesir.edu.tr/xmlui/handle/20.500.12462/9147>.

BICUDO, I. **Os elementos**. Editora Unesp, 2009. ISBN 9788571399358. Disponível em: <https://books.google.com.br/books?id=um94A66MDxkC>.

BLENDER. Noise texture node. **Blender 4.5 LTS Manual**, 2025. Disponível em: <https://docs.blender.org/manual/en/latest/compositing/types/texture/noise.html>.

____. Wavefront obj. **Blender 2.80 Reference Manual**, 2025. Disponível em: <https://docs.blender.org/manual/en/2.80/addons/io_scene_obj.html>.

BOURKE, P. Dla - diffusion limited aggregation. **Paraná Inteligência Artificial**, 1991. Disponível em: <https://paulbourke.net/fractals/dla/>.

BRAGA, F.; RIBEIRO, M. Diffusion limited aggregation: Algorithm optimization revisited. **Computer Physics Communications**, v. 182, n. 8, p. 1602–1605, 2011. ISSN 0010-4655. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0010465511001238>.

Chr. Clemens Lee. **Diffusion Limited Aggregation (DLA)**. 2006. Website. Disponível em: <http://www.kclee.de/clemens/java/dla/>.

CORDEIRO, M.; ZOLA, W. Construção paralela lock-free de octrees esparsas em gpu. In: **Anais da XXIV Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, RS, Brasil: SBC, 2024. p. 107–108. ISSN 2595-4164. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/28016>.

COSTA, P. C. da. Metodologia aplicada ao cálculo da dimensão fractal de formações urbanas utilizando o índice de desenvolvimento humano municipal (idhm) como critério de seleção. **Revista Mackenzie de Engenharia e Computação**, v. 14, n. 1, 2015. Disponível em: <https://editorarevistas.mackenzie.br/index.php/rmec/article/view/6472>.

COYNE, T. P. Performance portability of cuda across nvidia gpu architectures. **Virginia Tech**, 05 2025.

DUFF, C. M. The power of supercomputing applied to fractal image generation. **Digital Commons - Cal Poly Humboldt - Mathematics Senior Capstones**, 2023.

FAé, L.; GRIEBLER, D. Proposta de pipelines lineares de alto nível em rust utilizando gpu. In: **Anais da XXIV Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, RS, Brasil: SBC, 2024. p. 105–106. ISSN 2595-4164. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/28015>.

FENLASON, J.; STALLMAN, R. Gnu gprof. **O Sistema Operacional GNU**, 1998. Disponível em: <https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html>.

FILHO, S. L. M. A curva de koch (fractal floco de neve). **Paraná Inteligência Artificial**, 2002. Disponível em: <http://www.batebyte.pr.gov.br/Pagina/Curva-de-Koch-Fractal-Floco-de-Neve>.

FLYNN, M. Very high-speed computing systems. **Proceedings of the IEEE**, v. 54, n. 12, p. 1901–1909, 1966.

FRACTALIZE. **De onde veio o Conjunto de Mandelbrot?** 2020. Disponível em: <https://www2.ufjf.br/fractalize/2020/12/23/conjunto-de-mandelbrot/>.

FUZZO, R. A. **Fractais: Algumas Características e Propriedades**. Encontro de Produção Científica e Tecnológica, 2009. Disponível em: <https://www.fecilcam.br/nupem/anais_iv_epct/PDF/ciencias_exatas/10_FUZZO_REZENDE_SANTOS.pdf>.

GARCIA, F. *et al.* Coloring dynamical systems in the complex plane. **The University of the Basque Country**, 2008.

HANSEN, A. E. *et al.* Fractal particle trajectories in capillary waves: Imprint of wavelength. **Phys. Rev. Lett.**, American Physical Society, v. 79, p. 1845–1848, Sep 1997. Disponível em: <https://link.aps.org/doi/10.1103/PhysRevLett.79.1845>.

HAQUE, M. E. *et al.* **GPU Accelerated Fractal Image Compression for Medical Imaging in Parallel Computing Platform**. 2014. Disponível em: <https://arxiv.org/abs/1404.0774>.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Sixth Edition: A Quantitative Approach**. 6th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128119055.

HÖLSCHER, L. *et al.* Quantum-inspired fluid simulation of two-dimensional turbulence with gpu acceleration. **Phys. Rev. Res.**, American Physical Society, v. 7, p. 013112, Jan 2025. Disponível em: <https://link.aps.org/doi/10.1103/PhysRevResearch.7.013112>.

JURAEV, D. A.; MAMMADZADA, N. M. Fractals and its applications. **Karshi Multidisciplinary International Scientific Journal**, v. 1, n. 1, p. 25–38, 2024. Disponível em: <https://doi.org/10.22105/kmisj.v1i1.42>.

KIRK, D. B.; HWU, W.-m. W. **Programming Massively Parallel Processors: A Hands-on Approach**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN 0123814723.

LEONARCZYK, R.; GRIEBLER, D. Avaliação da auto-adaptação de micro-lote para aplicação de processamento de streaming em gpus. In: **Anais da XXIII Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, RS, Brasil: SBC, 2023. p. 123–124. ISSN 2595-4164. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/24523>.

LOC. **Wavefront OBJ File Format**. 2025. <https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml>.

MARTIN-GARIN, B. *et al.* Use of fractal dimensions to quantify coral shape. **Coral Reefs**, v. 26, p. 541–550, 09 2007.

MATHIGON. **O Conjunto de Mandelbrot**. 2023. Disponível em: <https://pt.mathigon.org/course/fractals/mandelbrot>.

MAYFIELD, W. D. *et al.* **Fractal Art Generation using GPUs**. 2016. Disponível em: <https://arxiv.org/abs/1611.03079>.

MEAKIN, P. Fractal structures. **Progress in Solid State Chemistry**, v. 20, n. 3, p. 135–233, 1990. ISSN 0079-6786. Disponível em: <https://www.sciencedirect.com/science/article/pii/007967869090001V>.

MEIR, E.; HALLER, B. Fractal mountain climbing. **MacTutor**, v. 7, n. 5, may 1991. Coluna "C Workshop". Disponível em: <https://benhaller.com/pubs/Meir1991.pdf>.

MENDONÇA, F. Aplicações da geometria fractal: uma proposta didática para o ensino médio. **Universidade de Alagoas**, 2016. Disponível em: <https://www.repositorio.ufal.br/bitstream/riufal/2412/1/Aplica%C3%A7%C3%B5es%20da%20geometria%20fractal_%20uma%20proposta%20did%C3%A1tica%20para%20o%20ensino%20m%C3%A9dio.pdf>.

NVIDIA. Cuda c++ programming guide. **NVIDIA Docs**, 2025.

____. Tuning cuda applications for nvidia ada gpu architecture. **NVIDIA Docs**, 2025. Disponível em: <https://docs.nvidia.com/cuda/ada-tuning-guide/index.html>.

OWENS, J. D. *et al.* Gpu computing. **Proceedings of the IEEE**, v. 96, n. 5, p. 879–899, 2008.

Programming Chaos. **Easily Program Coral-like Fractals with Diffusion Limited Aggregation**. 2023. Vídeo do YouTube. Disponível em: <https://www.youtube.com/watch?v=4_8a8JwXLp4>.

PURKIS, S. J. **Fractal Patterns of Coral Communities: Evidence from Remote Sensing (Arabian Gulf, Dubai, U.A.E.)**. Department of Marine and Environmental Sciences, 2006. Disponível em: <https://nsuworks.nova.edu/cgi/viewcontent.cgi?article=1048&context=occ_facpresentations>.

RAHMAD, M. H. *et al.* Comparison of cpu and gpu implementation of computing absolute difference. In: **2011 IEEE International Conference on Control System, Computing and Engineering**. [S.l.: s.n.], 2011. p. 132–137.

RUPP, K. **CPU, GPU and MIC Hardware Characteristics over Time**. 2016. Disponível em: <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>.

RYU, M.; BYEON, G.; KIM, K. **A GPU-Accelerated Distributed Algorithm for Optimal Power Flow in Distribution Systems**. 2025. Disponível em: <https://arxiv.org/abs/2501.08293>.

SALLOW, A. Implementation and analysis of fractals shapes using gpu-cuda model. **Academic Journal of Nawroz University**, v. 10, p. 1, 04 2021.

Samuel (sml). **Exploring Diffusion Limited Aggregation in Geometry Nodes**. 2025. Blender Community. Disponível em: <https://blenderartists.org/t/exploring-diffusion-limited-aggregation-in-geometry-nodes/1589322>.

SANTOS, B. P. dos. O problema do fractal de mandelbrot como comparativo de arquiteturas de memória compartilhada – gpu vs openmp. **Departamento de Ciências Exatas e Tecnológicas – Campus Soane Nazaré de Andrade - Universidade Estadual de Santa Cruz (UESC)**, 2011.

SANTOS, D. *et al.* AlgorÍtmos para estudos de limiar de percolaÇÃo atravÉs da dimensÃo fractal em hiper-redes. **HOLOS**, v. 5, p. 67, 11 2017.

SEDIVY, R. *et al.* Fractal analysis: An objective method for identifying atypical nuclei in dysplastic lesions of the cervix uteri. **Gynecologic Oncology**, v. 75, n. 1, p. 78–83, 1999. ISSN 0090-8258. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0090825899955166>.

SEI, N. Geometria fractal. **Bangladesh J. Bot.**, 2003. Disponível em: <https://www.mat.uc.pt/~mcag/FEA2003/GeometriaFractal.pdf>.

SIREGAR, V. *et al.* Efficient computation of mandelbrot set generation with compute unified device architecture (cuda). In: **2022 6th International Conference on Informatics and Computational Sciences, ICICoS 2022**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2022. (Proceedings - International Conference on Informatics and Computational Sciences), p. 1–5.

STALLINGS, W. **Arquitetura e Organização de Computadores**. Pearson Universidades, 2017. ISBN 9788543020532. Disponível em: <https://books.google.com.br/books?id=Yhpg0AEACAAJ>.

TANENBAUM, A. S. **Organização estruturada de computadores**. Pearson, 2013. Disponível em: <https://plataforma.bvirtual.com.br/Acervo/Publicacao/3825>.

THOMAS, W.; DARUWALA, R. D. Performance comparison of cpu and gpu on a discrete heterogeneous architecture. In: **2014 International Conference on Circuits, Systems, Communication and Information Technology Applications (CSCITA)**. [S.l.: s.n.], 2014. p. 271–276.

WITTEN, T. A.; SANDER, L. M. Diffusion-limited aggregation, a kinetic critical phenomenon. **Phys. Rev. Lett.**, American Physical Society, v. 47, p. 1400–1403, Nov 1981. Disponível em: <https://link.aps.org/doi/10.1103/PhysRevLett.47.1400>.

YANG, X.-J. **Modelling Fractal Waves on Shallow Water Surfaces via Local Fractional Korteweg-de Vries Equation**. Wiley Online Library, 2014. Disponível em: <https://doi.org/10.1155/2014/278672>.

YAO, W.; HU, L.; HOU, Y. A distributed parallel network intrusion detection system based on ray framework with gpu acceleration. **Concurrency and Computation: Practice and Experience**, v. 37, n. 9-11, p. e70021, 2025. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.70021>.

ZSAKI, A. M. Hardware-accelerated generation of 3d diffusion-limited aggregation structures. **Journal of Parallel and Distributed Computing**, v. 97, p. 24–34, 2016. ISSN 0743-7315. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0743731516300740>.

APÊNDICE A – DECLARAÇÃO DE USO DE INTELIGÊNCIA ARTIFICIAL

Durante a preparação deste trabalho, o autor utilizou as ferramentas: Google Gemini[1] versão 2.5 PRO para revisão ortográfica, ajustes de coesão textual e criação de um programa para automatizar a execução dos testes. Também foi usada a sua ferramenta integrada *Deep Research* para auxiliar na busca de artigos científicos. Após o uso destas ferramentas, o autor revisou e editou o conteúdo em conformidade com o método científico e assume total responsabilidade pelo conteúdo da publicação.

[1] https://gemini.google.com/