UNIVERSIdade FEEVALE

DOUGLAS NEVES SPINDLER

ALGORITMOS PARA GERAÇÃO DE FRACTAIS: UMA ANÁLISE DE DESEMPENHO DE MODELOS SEQUENCIAIS E PARALELOS baseados em ARQUITETURAS *MULTI-CORE* E *MANY-CORE*

Novo Hamburgo

2012

DOUGLAS NEVES SPINDLER

ALGORITMOS PARA GERAÇÃO DE FRACTAIS: UMA ANÁLISE DE DESEMPENHO DE MODELOS SEQUENCIAIS E PARALELOS baseados em ARQUITETURAS *MULTI-CORE* E *MANY-CORE*

Trabalho de Conclusão de Curso

apresentado como requisito parcial

à obtenção do grau de Bacharel

em Ciência da Computação pela

Universidade Feevale

Orientador: Gabriel da Silva Simões

Novo Hamburgo

2012DOUGLAS NEVES SPINDLER

Trabalho de Conclusão de Curso de Ciência da Computação, com título **ALGORITMOS PARA GERAÇÃO DE FRACTAIS: UMA ANÁLISE DE DESEMPENHO DE MODELOS SEQUENCIAIS E PARALELOS BASEADOS EM ARQUITETURAS *MULTI-CORE* E *MANY-CORE*,** submetido ao corpo docente da Universidade Feevale, como requisito necessário para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado por:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Prof. Me. Gabriel da Silva Simões**

**Orientador**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Banca Examinadora

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Banca Examinadora

Novo Hamburgo, junho de 2012.

Resumo

O advento dos processadores *multi-core* nos últimos anos é resultado da crescente demanda por poder de processamento pelos usuários. No segmento de placas gráficas, este fato tem resultado em GPUs com uma capacidade de processamento cada vez maior, proporcionada pela grande quantidade de unidades lógicas e aritméticas e de núcleos presentes dentro de cada processador, de modo que estes dispositivos tenham deixado de ser utilizados apenas para processamento gráfico e tenham se tornado ferramentas computacionais de propósito geral. Estes avanços na área de processamento paralelo têm viabilizado a resolução de diversos problemas no meio científico, entre eles a geração de fractais. No contexto dos sistemas dinâmicos, os algoritmos que descrevem a geração destes objetos são caracterizados pela aplicação, através de repetidas iterações, de uma determinada função complexa a um conjunto de dados de entrada, o qual, para os propósitos deste trabalho, consiste nos pixels de uma imagem. Dado que este processo pode ser feito de maneira independente para cada pixel, o mesmo apresenta um grande potencial de paralelização. Desta forma, o objetivo geral deste trabalho é prover uma análise de desempenho de algoritmos para geração de fractais baseados em uma arquitetura *many-core* com GPU e comparar os resultados obtidos com o desempenho dos mesmos algoritmos sendo executados de modo sequencial e paralelo com uma CPU *multi-core* tradicional. Para tanto, foram utilizados os modelos de programação OpenMP, para a implementação dos algoritmos paralelos baseados em CPU, e o modelo de programação e arquitetura CUDA, da NVIDIA, para as execuções com GPU. Os testes foram feitos utilizando seis diferentes funções para a geração dos fractais, e variando parâmetros como as dimensões da imagem e número máximo de iterações para cada uma delas. Os resultados comprovaram o ganho de desempenho esperado ao utilizar modelos baseados em GPU para a execução dos algoritmos, com alguns cenários apresentando *speedups* de quase duas ordens de magnitude em relação às execuções sequenciais. Entretanto, percebeu-se uma grande variação nos ganhos de desempenho obtidos para cada uma das funções, evidenciando uma forte dependência com as configurações dos parâmetros utilizados para os testes. As execuções com OpenMP apresentaram resultados bem mais modestos, em média aproximadamente três vezes mais rápidas que as execuções sequenciais, porém, com menor variação entre as diferentes funções.

Palavras-chave: Análise de desempenho. Fractais. Programação paralela. GPU. CUDA. OpenMP.

Abstract

The advent of multi-core processors in the last years is a result of the growing demand for computational power from the users. In the segment of graphics cards, this fact has resulted in GPUs with more and more computational capacity, provided by the great amount of arithmetic and logic units and cores inside each processor, in such a way that these devices are no longer used only for graphics processing, but have become general-purpose computing tools. These advances in the area of parallel programming have made feasible the resolution of many problems in the field of science, among which is the generation of fractals. In the context of dynamical systems, the algorithms which describe the generation of such objects are characterized by the application, through repeated iterations, of a given complex function to an input data set, which, for the purposes of this work, is represented by the pixels of an image. Given that this process can be done in an independent fashion for each of these pixels, it shows a great potential for parallelization. This way, the general objective of this work is to provide a performance analysis of fractal generation algorithms based on a many-core architecture with GPU and compare the results obtained with the performance of the same algorithms being run both in sequential and parallel fashions using a traditional multi-core CPU. For this, the OpenMP programming model was used for the implementation of the parallel algorithms based on CPU, whereas the CUDA programming model and architecture, from NVIDIA, was used for the executions with GPU. The tests were done using six different functions for the generation of fractals, varying parameters like the image dimensions and the maximum number of iterations for each of these functions. The results confirmed the expected performance gain by using GPU-based models for the execution of the algorithms, with some scenarios showing speedups of almost two orders of magnitude in comparison with the sequential runs. However, it was possible to notice a great variation in the performance gains obtained for each of the functions, making evident a strong dependency with the parameter setup used in the tests. The executions with OpenMP showed much more modest results, in average approximately three times faster than the sequential runs, but with a lower variation between the different functions.

Keywords: Performance analysis. Fractals. Parallel programming. GPU. CUDA. OpenMP.

Lista de Figuras

[Figura 1 – Representação fractal de uma samambaia (feto) (a) e de uma grama (b). 16](#_Toc327124649)

[Figura 2 – Transposição do plano complexo para coordenadas na tela. 19](#_Toc327124650)

[Figura 3 – O conjunto de Julia preenchido da função , para *c* = -0,52 + 0,57*i*. 20](#_Toc327124651)

[Figura 4 – O conjunto de Mandelbrot representando um “dicionário” de todos os conjuntos de Julia conectados possíveis. 23](#_Toc327124652)

[Figura 5 – Os períodos das órbitas dos pontos de cada bulbo do conjunto de Mandelbrot. 24](#_Toc327124653)

[Figura 6 – Conjunto de Julia em escala original (a) e ampliado (b). 25](#_Toc327124654)

[Figura 7 – O triângulo de Sierpinski (a) e a samambaia de Barnsley (b). 27](#_Toc327124655)

[Figura 8 – Um fractal do tipo *flame*. 29](#_Toc327124656)

[Figura 9 – O atrator de Lorenz no plano *x-z*. 30](#_Toc327124657)

[Figura 10 – Representação 3D de um conjunto de Julia gerado com quatérnios. 32](#_Toc327124658)

[Figura 11 – Exemplo de execução paralela de um laço do tipo *do*. 37](#_Toc327124659)

[Figura 12 – Comparativo entre os projetos de uma CPU e de uma GPU. 40](#_Toc327124660)

[Figura 13 – Modelo de processamento OpenCL 42](#_Toc327124661)

[Figura 14 – Relação entre os componentes em um ambiente AMD APP 43](#_Toc327124662)

[Figura 15 – Núcleo de um processador Larrabee 45](#_Toc327124663)

[Figura 16 – A hierarquia de *threads* na arquitetura CUDA 47](#_Toc327124664)

[Figura 17 – Exemplo de chamada de um *kernel*. 48](#_Toc327124665)

[Figura 18 – Modelo de memória de um dispositivo CUDA. 52](#_Toc327124666)

[Figura 19 – Organização de uma imagem normal e com a biblioteca CImg. 56](#_Toc327124667)

[Figura 20 – Fractais resultantes das funções 9 a 14. 58](#_Toc327124668)

[Figura 21 – Imagem gerada com limite de 20 (a) e 500 (b) iterações. 59](#_Toc327124669)

Lista de QUADROS

[Quadro 1 – Configuração de um IFS para um Triângulo de Sierpinski. 26](#_Toc327138691)

[Quadro 2 – Variações do algoritmo *Fractal Flame*. 28](#_Toc327138692)

[Quadro 3 – Palavras-chave para definição de um *kernel*. 48](#_Toc327138693)

[Quadro 4 – Características dos diferentes tipos de memória na arquitetura CUDA. 50](#_Toc327138694)

[Quadro 5 – Especificações técnicas da placa gráfica GeForce GT 330M. 55](#_Toc327138695)

[Quadro 6 – Médias de tempo obtidas para as execuções do primeiro teste no modo sequencial. 63](#_Toc327138696)

[Quadro 7 – Tempos obtidos para as execuções do segundo teste no modo sequencial. 65](#_Toc327138697)

[Quadro 8 – Resultados das execuções com OpenMP com *schedule static*, sem definir tamanho de *chunk*. 66](#_Toc327138698)

[Quadro 9 – Resultados das execuções com OpenMP com *schedule dynamic*, sem definir tamanho de *chunk*. 67](#_Toc327138699)

[Quadro 10 – Resultados das execuções com OpenMP com *schedule dynamic* e um *chunk* para cada *thread*. 68](#_Toc327138700)

[Quadro 11 – Resultados das execuções com OpenMP com *schedule dynamic* e *chunks* de tamanho DIM/32. 69](#_Toc327138701)

[Quadro 12 – Resultados das execuções com CUDA utilizando DIM x DIM blocos com uma *thread*. 70](#_Toc327138702)

[Quadro 13 – Resultados das execuções com CUDA utilizando DIM x DIM blocos com uma *thread* e cálculo de divergência utilizando a função *cuCabsf()*. 72](#_Toc327138703)

[Quadro 14 – Resultados das execuções com CUDA utilizando DIM/32 x DIM/16 blocos com 32 x 16 x 1 *threads*. 73](#_Toc327138704)

[Quadro 15 – Resultados das execuções com CUDA utilizando DIM/16 x DIM/16 blocos com 16 x 16 x 1 *threads*. 74](#_Toc327138705)

[Quadro 16 – Resultados das execuções com CUDA utilizando DIM/32 x DIM/32 blocos com 16 x 16 x 1 *threads* e cada uma processando 4 pixels. 75](#_Toc327138706)

LISTA DE GRÁFICOS

[Gráfico 1 – O aumento das dimensões da imagem causa um aumento quase linear no tempo de execução. 65](#_Toc327035944)

[Gráfico 2 – *Speedups* obtidos a partir das execuções com OpenMP em relação às execuções sequenciais. 77](#_Toc327035945)

[Gráfico 3 – *Speedups* obtidos a partir das execuções com CUDA em relação às execuções sequenciais. 78](#_Toc327035946)

[Gráfico 4 – Parcelas de tempo gastas com transferências de memória durante as execuções com CUDA. 80](#_Toc327035947)

Lista de Abreviaturas e Siglas

|  |  |
| --- | --- |
| AMD | Advanced Micro Devices |
| API | Application Programming Interface |
| APP | Accelerated Parallel Processing |
| CAL | Compute Abstraction Layer |
| CPU | Central Processing Unit |
| CUDA | Compute Unified Device Architecture |
| GPGPU | General-purpose Graphics Processing Unit |
| GPU | Graphics Processing Unit |
| IDE | Integrated Development Environment |
| IFS | Iterated Function System |
| MPI | Message Passing Interface |
| SDK | Software Development Kit |
| SIMD | Single Instruction, Multiple Data |
| SSE | Streaming SIMD Extension |
| ULA | Unidade Lógica e Aritmética |
| VAIO | Visual Audio Intelligent Organizer |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Sumário

[Introdução 11](#_Toc327035867)

[1 FRACTAIS 14](#_Toc327035868)

[1.1 APLICAÇÕES 15](#_Toc327035869)

[1.2 SISTEMAS DINÂMICOS 17](#_Toc327035870)

[1.3 ITERAÇÃO NO PLANO COMPLEXO 18](#_Toc327035871)

[1.4 OS CONJUNTOS DE JULIA 19](#_Toc327035872)

[1.4.1 Geração 20](#_Toc327035873)

[1.5 O CONJUNTO DE MANDELBROT 21](#_Toc327035874)

[1.6 OUTROS TIPOS DE FRACTAIS 25](#_Toc327035875)

[1.6.1 Sistemas de funções iterativas (IFS) 25](#_Toc327035876)

[1.6.2 *Fractal Flames* 27](#_Toc327035877)

[1.6.3 Atratores estranhos 29](#_Toc327035878)

[1.6.4 Quatérnios 30](#_Toc327035879)

[2 PROGRAMAÇÃO PARALELA 33](#_Toc327035880)

[2.1 MPI 34](#_Toc327035881)

[2.2 OPENMP 35](#_Toc327035882)

[2.3 DISCUSSÃO 38](#_Toc327035883)

[3 PROGRAMAÇÃO COM GPUS 40](#_Toc327035884)

[3.1 OPENCL 42](#_Toc327035885)

[3.2 AMD ACCELERATED PARALLEL PROCESSING (ATI STREAM) 43](#_Toc327035886)

[3.3 INTEL LARRABEE 44](#_Toc327035887)

[4 A Arquitetura CUDA 46](#_Toc327035888)

[4.1 *KERNELS* E HIERARQUIA DE *THREADS* 47](#_Toc327035889)

[4.2 *WARPS* 49](#_Toc327035890)

[4.3 MEMÓRIAS 49](#_Toc327035891)

[4.4 CAPACIDADE COMPUTACIONAL 52](#_Toc327035892)

[5 DESENVOLVIMENTO DOS algoritmos 54](#_Toc327035893)

[5.1 AMBIENTE DE DESENVOLVIMENTO E TESTES 54](#_Toc327035894)

[5.2 METODOLOGIA 56](#_Toc327035895)

[6 TESTES E análise dos resultados 62](#_Toc327035896)

[6.1 EXECUÇÕES EM MODO SEQUENCIAL 62](#_Toc327035897)

[6.2 EXECUÇÕES COM OPENMP 66](#_Toc327035898)

[6.3 EXECUÇÕES COM CUDA 70](#_Toc327035899)

[6.4 DISCUSSÃO 76](#_Toc327035900)

[CONCLUSÃO 82](#_Toc327035901)

[Referências Bibliográficas 85](#_Toc327035902)

[APÊNDICE 89](#_Toc327035903)

[APÊNDICE A – *KERNELS* UTILIZADOS PARA OS TESTES COM CUDA 89](#_Toc327035904)

[APÊNDICE B – FUNÇÃO SENOIDAL COMPLEXA PARA CUDA 92](#_Toc327035905)

[APÊNDICE C – ALGORITMOS PARA AS EXECUÇÕES SEQUENCIAIS E COM OPENMP 92](#_Toc327035906)

Introdução

A indústria de microprocessadores depara-se com uma crescente demanda por desempenho e pela capacidade de executar diversas tarefas simultaneamente por parte dos usuários, desenvolvedores e do meio científico. Contudo, limitações físicas dos componentes de *hardware* têm impossibilitado que a velocidade do *clock* dos processadores seja aumentada significativamente. Estes fatos fazem com que alternativas tenham de ser buscadas por parte dos fabricantes para suprir esta demanda por poder processamento das atuais aplicações.

Segundo Sanders e Kandrot (2011), o advento dos processadores *multi-core* nos últimos anos é resultado desta busca. Processadores com múltiplos núcleos já estão atualmente difundidos de tal modo que é inclusive difícil encontrar dispositivos com apenas um núcleo de processamento.

Especificamente no segmento de processadores gráficos, jogos com níveis de detalhes e realismo cada vez mais altos fizeram com que as GPUs se tornassem dispositivos com um imenso poder computacional. Estes dispositivos são atualmente providos com centenas de núcleos de processamento e proporcionam um alto nível de paralelismo para a execução das aplicações, caracterizando a arquitetura atualmente chamada de *many-core*. Assim, as GPUs começam a ser utilizadas não apenas como dispositivos voltados a processamento gráfico, mas como ferramentas computacionais de propósito geral (KIRK E HWU, 2010).

Esta capacidade de processamento dos dispositivos gráficos tem despertado o interesse de desenvolvedores em utilizá-los na resolução de problemas nas mais diversas áreas da ciência. Atualmente já existem estudos utilizando GPUs para a execução de algoritmos de simulações meteorológicas, dinâmica de fluidos, processamento de imagens médicas e *data mining*, dentre muitos outros (NVIDIA, 2012).

Um dos ramos da ciência beneficiados pelo uso de múltiplos processadores paralelos refere-se ao estudo dos fractais, objetos resultantes de repetidas iterações de uma função aplicadas sobre um determinado conjunto de dados. Dentre diversas aplicações dos fractais, a análise das imagens destes objetos é de especial interesse na área dos sistemas dinâmicos. Conforme Devaney (1990), a natureza complexa, as estruturas intricadas e o comportamento caótico exibidos nestas imagens, produzidas frequentemente através de funções simples, são estudados pelos cientistas na esperança de compreender o comportamento de sistemas mais complexos.

Embora diversas classificações e métodos para geração de fractais existam, alguns deles são descritos por algoritmos especialmente adequados para execuções paralelas, como é o caso dos conjuntos de Mandelbrot e de Julia, devido ao fato de que, nestes conjuntos, cada pixel da imagem pode ser processado de maneira completamente independente. Este fato, aliado ao grande volume de cálculos aritméticos necessários para exibir os detalhes destes objetos, faz com que a geração de fractais satisfaça perfeitamente os requisitos para fazer uso máximo do poder computacional das GPUs. A análise de desempenho destes algoritmos utilizando modelos paralelos de programação caracteriza o foco deste trabalho.

A escolha do tema deu-se com base no interesse do autor por fractais, especialmente no que diz respeito ao nível de detalhes das imagens, às formas de aparência exótica e à fundamentação matemática que envolve esta área de estudo. Este interesse foi unido à percepção de um crescente uso de modelos de programação que façam uso de dispositivos gráficos como ferramentas computacionais de propósito geral, em especial a arquitetura CUDA, sendo aplicados a problemas nos mais diversos ramos da ciência, motivando a pesquisa sobre trabalhos existentes que correlacionassem estas duas áreas. Constatou-se que, apesar de existirem diversas referências e exemplos de algoritmos na literatura sobre geração de fractais baseados em algoritmos paralelos, incluindo materiais recentes já abordando o tema com o uso de GPUs, existe uma carência de estudos que apresentem análises de desempenho e comparativos entre os resultados obtidos a partir da execução destes algoritmos com modelos baseados em GPUs e processadores tradicionais. Além disso, o fato de julgar que os dois temas – geração de fractais e programação paralela e com GPUs – unem estudos nos âmbitos da ciência e da computação consistiu em uma motivação adicional a este estudo, de modo que sejam plenamente pertinentes ao curso do autor. Assim, surgiu o problema abordado neste trabalho: qual o ganho de desempenho que pode ser alcançado ao se utilizar GPUs para a geração de fractais, em comparação com modelos de programação tradicionais baseados em CPUs?

Desta forma, o objetivo geral deste estudo é verificar o ganho de desempenho alcançado para algoritmos para geração de fractais executados em uma arquitetura *many-core*, neste caso utilizando uma GPU através do modelo de programação CUDA, comparando os resultados obtidos com o desempenho dos mesmos algoritmos sendo executados de forma sequencial e paralela em uma CPU *multi-core*, utilizando o modelo OpenMP. Para alcançar este objetivo geral foram definidos os seguintes objetivos específicos: a realização de uma pesquisa bibliográfica sobre fractais, programação paralela e modelos de programação utilizando GPUs como ferramentas computacionais de propósito geral; o desenvolvimento de algoritmos para geração de fractais utilizando os três modelos de implementação; a avaliação dos resultados obtidos através destes modelos; e propostas de trabalhos futuros relacionados à área deste estudo.

O presente trabalho está dividido em seis capítulos. O primeiro deles refere-se à fundamentação teórica sobre fractais, com referências a autores como Barnsley, Falconer, Mandelbrot e Devaney. Neste capítulo são descritas as características de tais objetos, suas aplicações e alguns exemplos de fractais famosos na literatura, com ênfase nos conjuntos de Mandelbrot e de Julia, escolhidos para o desenvolvimento dos algoritmos utilizados para as análises deste estudo. O segundo capítulo aborda conceitos de programação paralela, destacando duas das principais abordagens utilizadas atualmente: o modelo baseado em troca de mensagens MPI, e um especial enfoque ao modelo OpenMP, citando autores como Dongarra, Wilkinson, Allen e Chandra. O capítulo 3 traz o conceito de GPUs como ferramentas computacionais de propósito geral. São abordados o modelo OpenCL e as arquiteturas AMD Accelerated Parallel Processing e Intel Larrabee, alternativas à arquitetura CUDA, da NVIDIA, escolhida como foco deste trabalho. Esta arquitetura é detalhada no capítulo 4, sendo conceituados tópicos como *kernels*, hierarquia de *threads* e os diferentes tipos de memória disponibilizados, através de referências a autores como Kirk, Hwu, Sanders, Kandrot e Farber, além de documentos disponibilizados pela própria NVIDIA. Encerrada a fundamentação teórica, são detalhados o desenvolvimento dos algoritmos e metodologia, no capítulo 5 e apresentados os testes e as análises dos resultados no capítulo 6.

Por fim, é apresentada a conclusão deste estudo. Nela são destacados os resultados mais relevantes obtidos a partir da execução dos testes e apontadas propostas de futuros trabalhos.

# FRACTAIS

Historicamente, o interesse pela geometria tem sido estimulado por suas aplicações na natureza. A elipse, por exemplo, assumiu importância como a forma das órbitas planetárias, bem como a esfera foi utilizada para representar o formato da Terra. Mesmo sabendo que órbitas não são exatamente elípticas e nem que a Terra é exatamente esférica, estas aproximações podem ser perfeitamente adequadas para descrever muitos objetos.

Entretanto, muitos padrões encontrados na natureza são tão irregulares e com nível de complexidade tão alto que não podem ser adequadamente descritos pela tradicional geometria euclidiana. Conforme mencionado por Mandelbrot (1982, p. 1), o criador do termo fractal, “nuvens não são cones, litorais não são círculos e nem o raio viaja em uma linha reta”. Assim, Mandelbrot propôs uma nova geometria da natureza, a geometria fractal, de modo a prover melhores representações de padrões irregulares e fragmentados – a própria origem do termo fractal, baseada no adjetivo latino *fractus*, significa “fragmentado” ou “irregular”. Falconer (1990, p. xiii) endossa a motivação para tal proposta afirmando que “conjuntos irregulares provêm uma representação muito melhor de muitos fenômenos naturais do que as figuras da geometria clássica”. Assim, a geometria fractal compõe um *framework* para o estudo de tais conjuntos irregulares.

Entretanto, não existe um consenso a respeito da definição exata de um fractal. Para Devaney (1990, p. 130), por exemplo, “um fractal é, basicamente, uma forma geométrica que apresenta duas propriedades especiais: o objeto é autossimilar e possui dimensão fracionária”.

Falconer (1990, p. xx), entretanto, defende que

a definição de fractal deveria ser considerada da mesma maneira que um biólogo define ‘vida’. Não há uma definição rígida e rápida, mas apenas uma lista de propriedades características de algo vivo, como a habilidade de se reproduzir, ou de se mover, ou ainda de existir, até certo ponto, independentemente do ambiente. A maior parte das coisas vivas possui a maioria das características da lista, embora haja objetos vivos que sejam exceções a cada uma delas. Do mesmo modo, parece melhor considerar um fractal como um conjunto que tem propriedades como as listadas abaixo, ao invés de procurar por uma definição que irá quase certamente excluir alguns casos interessantes.

Ainda conforme Falconer (1990), tais propriedades, para um dado conjunto fractal *F*, descrevem que:

* *F* possui uma estrutura fina, com detalhes em escalas arbitrariamente pequenas;
* *F* é irregular demais para ser descrito em linguagem geométrica tradicional, tanto localmente quanto globalmente;
* *F* tem, muitas vezes, alguma forma de autossimilaridade, talvez aproximada ou estatística;
* Geralmente a dimensão fractal de *F* é maior que sua dimensão topológica;
* Na maioria dos casos de interesse *F* é definido de modo muito simples, por exemplo, recursivamente.

Barnsley (1993) descreve o estudo da geometria fractal como uma extensão à geometria clássica, sendo uma nova maneira científica de analisar nuvens, florestas ou o arranjo das penas nas asas de um pássaro, que pode ser usada para construir modelos precisos de estruturas físicas.

## APLICAÇÕES

Como mencionado no início do capítulo, a motivação para o estudo da geometria fractal é tentar prover uma representação mais apurada de elementos da natureza. Alguns destes elementos, inicialmente citados por Mandelbrot (1982), incluem litorais, montanhas, árvores, folhas, flocos de neve, raízes e relâmpagos. Na área da astronomia também são utilizados fractais para a representação de galáxias ou crateras lunares.

Entretanto, a representação de objetos da natureza não caracteriza a única aplicação dos fractais. Falconer (1990) cita ainda a compressão de dados, de modo a representar imagens complexas utilizando apenas uma pequena quantidade de informações; a geração de paisagens por computador através de superfícies Brownianas; ou ainda simulações de dinâmicas de fluidos.

Na economia, modelos fractais podem ser utilizados para analisar o mercado. Mandelbrot (1997, p. 28) defende tal abordagem devido à “possibilidade de identificar estacionariedade e escala como princípios de invariância na economia” e que tais características (1997, p. 31), “implementadas adequadamente e com a ajuda de gráficos computacionais, são suficientes para representar uma enorme riqueza de comportamentos complexos”.

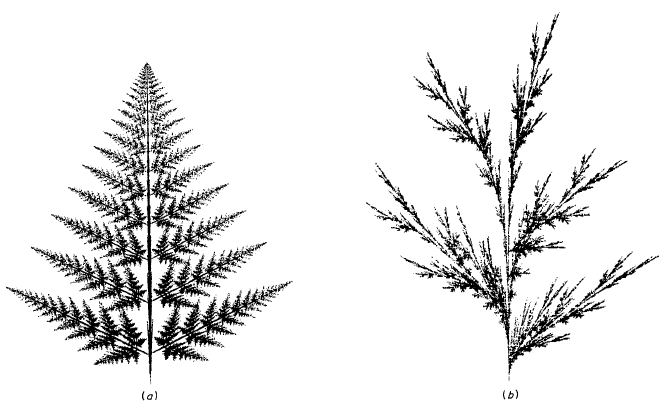


Figura – Representação fractal de uma samambaia (feto) (a) e de uma grama (b).

Fonte: Falconer (1990, p. 133).

As artes representam outra aplicação dos fractais. Devido às complexas e intricadas imagens que podem gerar, estes objetos podem ser utilizados com propósito decorativo, uma vez que sejam escolhidas cores apropriadas. Mitchell (1999) afirma que a arte fractal é em muitos aspectos similar à fotografia, já que o uso das luzes, sombras e movimentos são elementos também considerados por um artista fractal. Ainda, caracteriza esta área como sendo uma subclasse das artes visuais em duas dimensões, visto que ambos segmentos compartilham de elementos como cores, composição, equilíbrio, entre outros, e afirma que o trabalho desempenhado pelo artista fractal requer esforço e inteligência para manipular adequadamente todos os elementos que compõem a imagem.

Fractais também são utilizados na música. Aplicações nesta área são possíveis devido ao fato de que pode-se gerar padrões de sons a partir de estruturas geométricas. Brothers (2011) afirma que, assim como nas imagens, a música pode exibir uma grande variedade de comportamentos escalares, tendo identificado em suas pesquisas características como a autossimilaridade em relação à duração, altura, intervalo, tema e estrutura.

Xiao (2005) aponta ainda aplicações em compressão e codificação de áudio, utilizando algoritmos similares aos de compressão fractal de imagens. O autor afirma (2005, p. 19) que “tecnicamente, a diferença é o modo de representar os blocos: um bloco de imagem é representado como uma matriz de pixels, enquanto um bloco de áudio é representado por um vetor de valores de amostragem”. Ainda, afirma que as principais vantagens da codificação fractal são a simplicidade a habilidade de alcançar altas taxas de compressão. Entretanto, resultados satisfatórios nem sempre são atingidos, visto que a qualidade da compressão depende diretamente do número de padrões e autossimilaridades encontradas. Além disso, o processo de codificação pode consumir muito tempo devido à busca e comparação dos blocos.

Muitas outras aplicações de modelos fractais são possíveis, sendo as citadas acima apenas exemplos para contextualizar a importância do estudo dessa área. Não é o foco deste trabalho detalhar todas estas aplicações. Os leitores podem referir a Frame, Mandelbrot e Neger (2011) para uma extensa lista de outras possíveis aplicações. Entretanto, pode-se perceber, através dos exemplos de aplicações citados, que os fractais são estudados em uma extensa gama de áreas da ciência e que, por isso, justifica-se pensar no custo computacional para a geração de tais representações.

As próximas seções são destinadas a apresentar alguns tipos de fractais famosos na literatura. É possível agrupar estes objetos em classificações com base em alguns critérios como, por exemplo, o método para geração destes ou o nível de autossimilaridade à medida que se aumenta a escala. Um foco especial será dado aos fractais utilizados para o desenvolvimento deste trabalho: os conjuntos de Julia e o conjunto de Mandelbrot.

Para que se possa conceituar estes conjuntos, faz-se necessário, antes, comentar a respeito de sistemas dinâmicos, a área da matemática que engloba tais conjuntos, e sobre iterações utilizando variáveis complexas.

SISTEMAS DINÂMICOS

Os sistemas dinâmicos são definidos por Becker e Dörfler (1989) como sendo a área da matemática dedicada ao estudo de sistemas sujeitos a mudanças e cujo comportamento é ditado através de iterações. Estas iterações consistem em aplicar uma função matemática repetidas vezes sobre um conjunto de dados, utilizando o resultado da função anterior como valor de entrada para a atual. Os sistemas dinâmicos que nos cercam, como, por exemplo, o tempo, a economia global ou populações biológicas, são compostos de um número tão grande de parâmetros e mudam tão rapidamente que a tarefa de analisar seus comportamentos é extremamente difícil.

Entretanto, Devaney (1990) comenta que mesmo sistemas muito simples, até mesmo dependentes de apenas uma variável, podem se comportar de modo tão imprevisível quanto o mercado de ações, por exemplo. O fato de que esta imprevisibilidade, chamada de caos, pode ocorrer nos sistemas mais simples permite que os cientistas os analisem na esperança de compreender o comportamento de sistemas mais complexos, como a meteorologia ou sistemas econômicos. Este comportamento caótico ocorre até mesmo em funções quadráticas, quando estas são tratadas como sistemas dinâmicos. Nestes casos, o conjunto de números que geram tal comportamento caótico é chamado de conjunto de Julia, inicialmente estudado pelo matemático francês Gaston Julia.

ITERAÇÃO NO PLANO COMPLEXO

Os conjuntos de Julia são um tópico de estudo da área de dinâmica complexa, área esta que estuda o comportamento de sistemas dinâmicos definidos por funções que utilizam números complexos. Reade (2003) menciona que os números complexos surgiram do desejo de se extrair a raiz quadrada de um número negativo, motivado, segundo Burton (2011), de modo a satisfazer a necessidade de que todas as equações quadráticas e cúbicas possuíssem soluções. Estes números são representados na forma , onde *x* compõe a parte real do número, e *y* a parte imaginária, com *x* e *y* pertencendo ao conjunto dos números reais, e *i* satisfazendo a condição .

O estudo destes sistemas através da utilização de números complexos apresenta uma característica interessante, conforme apontado por Devaney (1990). O autor menciona que uma razão para a utilidade destes números é que os mesmos podem ser representados em um plano bidimensional de modo natural, com traçado no ponto com coordenadas (*x*, *y*). Assim, pode-se facilmente transpor um número complexo para um pixel em uma imagem, dado que um único número representa dois valores. Tal transposição é ilustrada na Figura 2 a seguir:

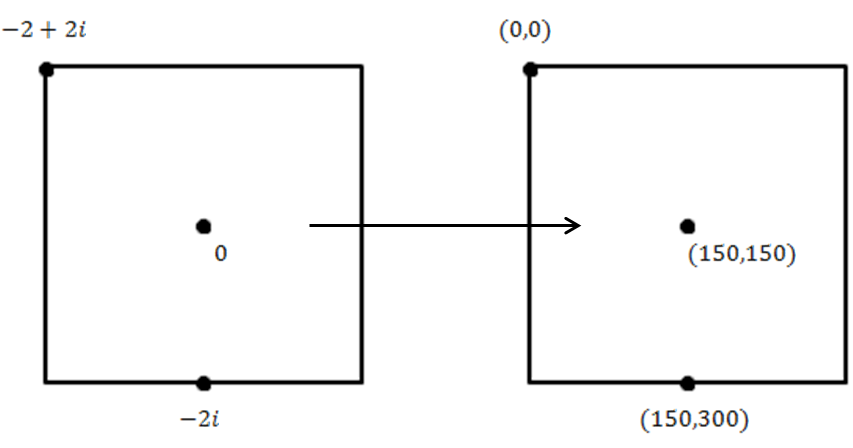


Figura – Transposição do plano complexo para coordenadas na tela.

Fonte: Adaptado de Devaney (1990, p. 80).

OS CONJUNTOS DE JULIA

Os conjuntos de Julia provêm algumas das mais impressionantes ilustrações de como um processo aparentemente simples pode gerar conjuntos altamente intricados. Funções no plano complexo C tão simples quanto , sendo *c* uma constante, podem criar fractais de aparência exótica. Tais conjuntos surgem juntamente com a iteração de uma função de uma variável complexa (FALCONER, 1990).

Uma definição mais formal é dada por Addison. O autor (1997) define que um conjunto de Julia de um polinômio é composto por todos os pontos situados entre os valores cujas órbitas convergem ao centro do atrator – valores estes que compõem o chamado “conjunto prisioneiro” ou “conjunto de Julia preenchido”– e os pontos que escapam ao infinito. Isto significa que um ponto no conjunto de Julia possui órbita que não escapa ao infinito, mas pontos próximos deste possuem órbitas que o fazem.

Quando é possível alcançar um ponto no conjunto de Julia a partir de qualquer outro ponto que também pertença ao conjunto, sem sair do mesmo, diz-se que este conjunto é conectado. Quando isto não é possível, é dito que o conjunto é desconectado (ADDISON, 1997). Ainda, Devaney (2006) comenta que os conjuntos de Julia são obrigatoriamente ou conectados, sendo formados por uma única peça, ou então formados por infinitos pontos separados. Não existe um meio-termo, sendo impossível dizer que um conjunto seja composto por um número finito de peças. Estes conceitos serão especialmente importantes quando forem abordados o conjunto de Mandelbrot.

Geração

Dada a natureza iterativa de um sistema dinâmico, descrita no início desta seção, e considerando que a tela pode representar um plano complexo bidimensional, com cada pixel representando uma coordenada, pode-se pensar na geração de conjuntos de Julia de modo que uma função complexa seja aplicada repetidamente sobre cada ponto de uma imagem.

Os conjuntos mais comuns são aqueles gerados a partir de funções quadráticas da forma , sendo *c* uma constante, definida por Addison (1997) como o parâmetro de controle da função. Cada diferente valor de *c* produz um conjunto de Julia diferente como resultado. Um exemplo é mostrado na Figura 3 abaixo:

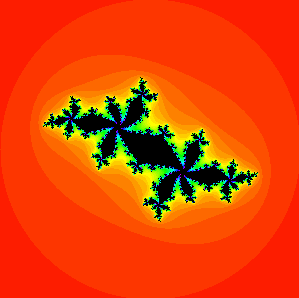


Figura – O conjunto de Julia preenchido da função , para *c* = -0,52 + 0,57*i*.

Fonte: Devaney (2006).

Devaney (1990) comenta que um dos motivos para a escolha de funções desta forma é que as funções quadráticas compõem a mais simples família de funções não-lineares no plano complexo que exibem estruturas matemáticas ricas em detalhes. Entretanto, é possível generalizar a definição de tais funções para a forma , de modo a utilizar qualquer função (BOURKE, 2001). Outras famílias de funções comumente utilizadas, conforme Devaney (1990), incluem polinômios de grau maior que 2 e funções transcendentais, como exponenciais, logarítmicas e trigonométricas.

O princípio básico da geração de conjuntos de Julia, descrito por Bourke (2001), consiste em mapear os pixels da imagem a uma região do plano complexo. Cada pixel representa o valor inicial da série de iterações da função complexa a ser aplicada. A função é então aplicada sobre cada pixel e este é colorido diferentemente caso divirja ao infinito ou não. É possível ainda colorir os pixels de acordo com o número de iterações necessárias para identificar que um ponto diverge ao infinito, o que provê uma visualização mais realista da dinâmica do sistema (DEVANEY, 1990).

Bourke (2001) cita que o critério de divergência, no entanto, nem sempre é trivial, de modo que em alguns casos precisa-se de um número muito alto de iterações para determinar se um ponto diverge ou não. Assim, tipicamente assume-se que a série tende ao infinito caso um determinado valor seja excedido. Do mesmo modo, caso após um número máximo de iterações o valor não tenha excedido o limite, assume-se que o ponto pertença ao conjunto. Este algoritmo é conhecido como algoritmo de tempo de escape (do inglês *escape-time algorithm*). Para funções na forma , o critério de escape consiste em testar se . Os detalhes a respeito das provas matemáticas para esta condição podem ser encontrados em Frame, Mandelbrot e Neger (2011) e Devaney (1990).

Algumas limitações surgem com esta abordagem. Bourke (2001) explica que ao definir um número máximo de iterações para verificar se a série diverge ao infinito ou não, eventualmente o algoritmo incluirá alguns valores no conjunto que na verdade não pertencem ao mesmo. Ao mesmo tempo, o algoritmo naturalmente ficará mais lento ao definir-se um número maior de iterações. Deve-se, portanto, buscar um número de iterações que produza uma imagem de qualidade aceitável em um tempo viável. Frame, Mandelbrot e Neger (2011) ainda comentam que, visto que uma tela é composta por uma matriz de pixels, na prática é possível apenas utilizar um mapeamento finito do plano complexo para os valores de *z*.

O CONJUNTO DE MANDELBROT

O conjunto de Mandelbrot é provavelmente o fractal mais famoso na matemática. Frame, Mandelbrot e Neger (2011) afirmam que o conjunto desafia noções familiares de simplicidade e de complexidade, visto que uma fórmula simples, envolvendo apenas multiplicação e adição, produz uma forma geométrica de grande beleza orgânica e variação infinitamente sutil. Além disso, afirmam que muito do interesse por dinâmica complexa foi motivado por esforços para entender as imagens do conjunto de Mandelbrot. O conjunto leva o nome do matemático Benoît Mandelbrot, um dos primeiros a estudá-lo.

Este conjunto é composto por todos os valores de *c* para os quais a órbita correspondente de 0 para a função não escapa ao infinito (DEVANEY, 2006). Deste modo, diferentemente dos conjuntos de Julia, onde os pixels são mapeados aos valores iniciais de *z* e utiliza-se um valor fixo no parâmetro de controle *c*, o conjunto de Mandelbrot é gerado mapeando os pixels aos valores de *c*, e iniciando a série de iterações com *z* igual a 0. Devaney (2006) resume dizendo que o conjunto de Mandelbrot é gerado no plano *c*, ou plano de parâmetros, e ilustra a órbita de 0 para todos os valores possíveis de *c*, enquanto o conjunto de Julia é uma figura no plano *z*, ou plano dinâmico, que ilustra todas as órbitas de para um valor fixo de *c*. Deste modo, a geração do conjunto de Mandelbrot pode ser feita utilizando o mesmo algoritmo descrito na seção anterior para os conjuntos de Julia, apenas mapeando-se os pixels da tela para valores de *c*, e não de *z*.

As semelhanças entre estes conjuntos não são coincidentais, e uma relação direta pode ser estabelecida entre os mesmos. Peitgen e Richter (1986) definem o conjunto de Mandelbrot como o princípio que indica o tipo de um dado conjunto de Julia. Um conjunto de Julia é uma estrutura conectada caso um ponto *c* no plano complexo pertença ao conjunto de Mandelbrot, ou quebrada em infinitas partes caso não pertença, gerando uma nuvem de pontos chamada de Poeira de Fatou. Este fato faz com que seja de especial interesse a análise da borda do conjunto de Mandelbrot, visto que é ali que os conjuntos de Julia correspondentes mudam sua natureza. Ao examinar esta borda é possível perceber o quão intricados e irregulares são estes conjuntos. Além disso, através de sucessivas ampliações da imagem nesta região do conjunto é possível perceber que o padrão inicial é replicado, características estas que ilustram a natureza fractal do objeto.

Dado que a posição de um ponto no conjunto de Mandelbrot dita o comportamento dos conjuntos de Julia, diz-se que o primeiro serve como um “dicionário” de todos os valores de *c* que produzem conjuntos de Julia conectados (ADDISON, 1997). Tal conceito é demonstrado na Figura 4. Todos os valores de *c* presentes na região em preto produzem conjuntos de Julia conectados:

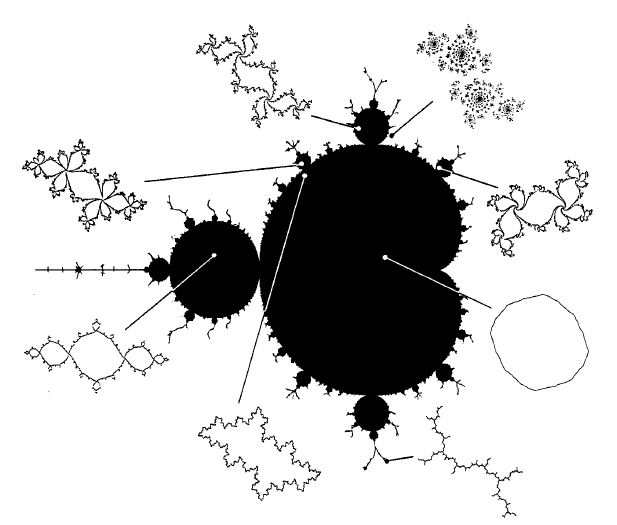


Figura – O conjunto de Mandelbrot representando um “dicionário” de todos os conjuntos de Julia conectados possíveis.

Fonte: Falconer (1990, p. 213).

Uma grande quantidade de informações pode ser extraída a partir de uma análise matemática do conjunto de Mandelbrot, segundo Devaney (2006). Os bulbos anexados à figura central de forma cardioide, por exemplo, indicam o período[[1]](#footnote-1) da órbita dos pontos do conjunto. Desta forma, todos os pontos dentro de um bulbo de período 2 consequentemente possuem um ciclo de atração com período 2. A Figura 5 ilustra esta relação:

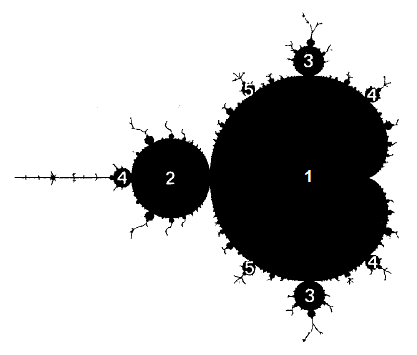


Figura – Os períodos das órbitas dos pontos de cada bulbo do conjunto de Mandelbrot.

Fonte: Adaptado de Falconer (1990, p. 216).

Os períodos de atração de cada ponto pertencente ao conjunto possuem a mesma relação com as formas similares a antenas que emanam dos bulbos e cardioide central. Cada antena possui um número de ramificações correspondente ao período do bulbo nas quais se encontram. Ainda, dada a relação entre o conjunto de Mandelbrot e os conjuntos de Julia descrita anteriormente, o conjunto de Julia produzido a partir de um valor de c terá infinitos pontos de junção unindo tantas formas quanto o valor do período de c. Este conceito é melhor visualizado através da Figura 6, tendo um fractal em escala original (a) e um ampliado (b). Os infinitos pontos de junção em (b) sempre unem três partes da região em preto, denotando um fractal gerado a partir de um valor de c presente em um bulbo de período igual a três no conjunto de Mandelbrot:

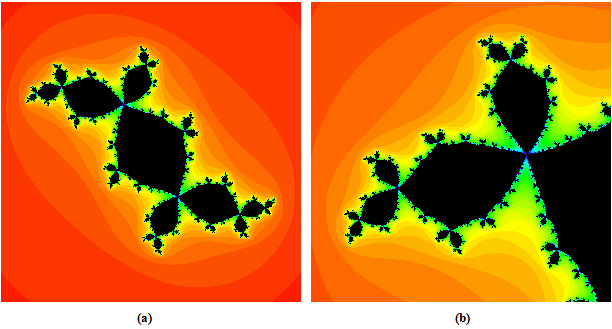


Figura – Conjunto de Julia em escala original (a) e ampliado (b).

Fonte: Devaney (2006).

Da mesma forma que ocorre com os conjuntos de Julia, é possível generalizar a definição do conjunto de Mandelbrot para utilizar qualquer função complexa. Ainda, variações análogas do conjunto são possíveis. White (2009) descreve variações tridimensionais, inclusive com polinômios hipercomplexos, denominadas Mandelbulbs, enquanto Green (2012) detalha um método alternativo para a exibição do conjunto de Mandelbrot tradicional, chamado de Buddhabrot.

OUTROS TIPOS DE FRACTAIS

Além dos tipos descritos nas seções anteriores, a literatura dispõe de diversos outros estudos de diferentes aplicações e métodos para geração de fractais. Nesta seção são apresentados alguns destes modelos, os quais podem ser aprofundados em futuros trabalhos.

Sistemas de funções iterativas (IFS)

Os sistemas de funções iterativas (do inglês *iterated function systems*), ou IFS, compõem uma classe de fractais inicialmente estudada por Barnsley, onde os objetos gerados sempre são autossimilares.

Barnsley, Hutchinson e Stenflo ([ca. 2004], p. 2) citam que

os IFS provêm modelos para certas plantas, folhas e samambaias (fetos) em virtude da autossimilaridade que frequentemente ocorre em estruturas ramificadas na natureza. Mas a natureza também exibe aleatoriedade e variação de um nível a outro, de modo que duas samambaias nunca são exatamente iguais, e frondes ramificadas transformam-se em folhas em uma escala menor.

A geração de um IFS consiste em iterar uma série de transformações, geralmente afins, a um determinado conjunto inicial. Estas transformações afins são utilizadas para representar parâmetros como, por exemplo, rotação, translação ou escala, de modo que à medida que vão sendo iterativamente aplicadas, réplicas do padrão original são construídas. Draves e Reckase (2003) mencionam que as funções geralmente são contrativas, de maneira a garantir que os pontos convirjam. Isto faz com que as cópias geradas sejam réplicas diminuídas do conjunto original.

Os dois métodos tradicionais de geração de um IFS, definidos por Barnsley (1993) são os algoritmos determinístico e de iteração aleatória, chamado de “jogo do caos” (do inglês *chaos game*). O primeiro consiste em computar uma sequência de conjuntos a partir de um conjunto inicial, que pode ser um ponto ou uma forma. Para tanto, são definidos os parâmetros do IFS em uma matriz como a ilustrada abaixo:



Quadro – Configuração de um IFS para um Triângulo de Sierpinski.

Fonte: Barnsley (1993, p. 85).

Onde *w* identifica a transformação afim. O parâmetro *p* é utilizado apenas no método de iteração aleatória. Estes valores são aplicados sucessivamente aos conjuntos a partir de um conjunto inicial *A*0, de modo a satisfazer o sistema abaixo:

()

O método do jogo do caos, por sua vez, baseia-se em selecionar aleatoriamente um ponto no plano e aplicar iterativamente uma das transformações afins do IFS, escolhida também aleatoriamente, mas de acordo com a probabilidade definida no parâmetro *p*, de modo que . À medida que os pontos resultantes das transformações são desenhados, a figura vai convergindo ao atrator do IFS. Quanto maior o número de iterações realizadas, maior a qualidade da imagem resultante.

O IFS descrito acima define o triângulo de Sierpinski, um dos fractais mais famosos na literatura. A mostra este fractal, juntamente com a samambaia (ou feto) de Barnsley, outro exemplo famoso de IFS, para os quais as cores identificam as regiões da imagem geradas a partir das diferentes transformações afins:

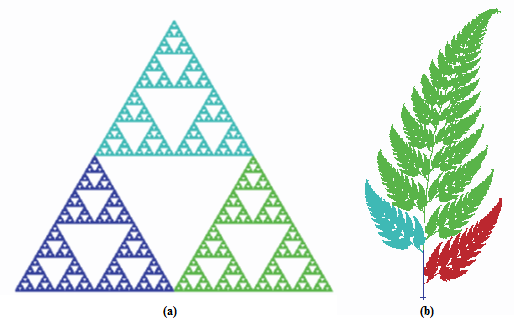


Figura – O triângulo de Sierpinski (a) e a samambaia de Barnsley (b).

Fonte: Helmberg (2007, p. 42 e 99).

*Fractal Flames*

*Fractal Flames* é o nome dado a uma extensão da classe de algoritmos IFS, desenvolvida por Draves e Reckase (2003). Esta classe difere dos IFS tradicionais principalmente pelo fato de usar funções não-lineares. Os autores (2003) comentam que um IFS normalmente utiliza uma função linear na forma

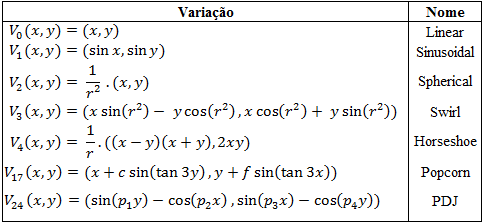
()

onde os valores *a* até *f* são coeficientes que definem o comportamento da função, de modo que cada combinação caracteriza uma imagem diferente como resultado. Um fractal do tipo flame, por sua vez, é gerado modificando-se a função descrita acima:

()

Onde:

* é uma função de pós-transformação, descrita da mesma forma da função (1), com a possibilidade de utilizar coeficientes diferentes;
* é uma matriz de coeficientes de mistura (*blending*); e
* é uma função denominada variação. Cada variação altera a forma e característica da imagem de maneiras específicas. Algumas destas variações são exibidas no Quadro 2:



Quadro – Variações do algoritmo *Fractal Flame*.

Fonte: Draves e Reckase (2003, p. 5).

Onde

()

.

Os autores (2003) classificam a variação como dependente, pois ela depende de dois coeficientes, *c* e *f*. A variação é paramétrica, pois ela necessita de quatro parâmetros externos (*, , ,* ) para descrever seu comportamento. São descritas, no total, 49 variações diferentes.

Para extrair informações sobre quais partes do atrator IFS são geradas a partir de quais funções, o algoritmo faz uso de cores, associadas a cada uma das variações. Ainda, o algoritmo flame ainda difere dos IFS normais por utilizar um histograma para construir a imagem, o que permite utilizar um mapa de cores representando diferentes densidades, com cores diferentes. Estas duas características adicionam propriedades estéticas interessantes à imagem final, produzindo efeitos de movimento e criando uma aparência 3D, mesmo que a imagem seja renderizada em 2D.



Figura – Um fractal do tipo *flame*.

Fonte: Draves e Reckase (2003, p. 2).

Atratores estranhos

Um atrator é definido por Addison (1997) como sendo o conjunto de pontos dos quais a órbita de um sistema dinâmico se aproxima à medida que o número de iterações aumenta ao infinito. Quando este atrator possui uma órbita não periódica, ou seja, uma órbita que possui um comportamento caótico, ele é chamado de atrator estranho (*strange attractor*). O autor (1997) ainda comenta que uma propriedade chave destas estruturas é que elas possuem dimensão fractal. Para Yan et al. (2010), nos casos de sistemas em que há uma dependência sensível às condições iniciais do atrator, de modo que trajetórias que iniciem muito próximas eventualmente divirjam, o resultado é um atrator estranho. Esta é para Sprott (1993), inclusive, a principal característica do caos, e refere-se ainda a tais sistemas como atratores caóticos.

Um dos exemplos mais famosos de atratores estranhos é o chamado atrator de Lorenz. Sprott (1993) comenta que Lorenz, ao estudar modelos de convecção atmosférica, descobriu um sistema sensivelmente dependente das condições iniciais, cujo fluxo é ditado pelas três equações abaixo:

()

()

()

Onde *σ* é o número de Prandtl e *ρ* é o número de Rayleigh. Ao definir os valores *σ* = 10, *β* = 8/3 e *ρ* = 28, o sistema exibe um comportamento caótico. Esta dependência das condições iniciais deu origem ao termo “efeito borboleta”. Ao imprimir os pontos gerados pela trajetória do atrator à medida que o número de iterações cresce, obtém-se a imagem ilustrada na :

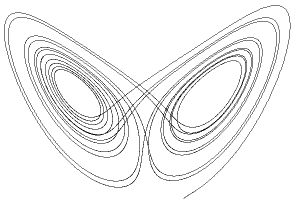


Figura – O atrator de Lorenz no plano *x-z*.

Fonte: Bradley (2010).

Outros exemplos famosos de tais sistemas incluem ainda o atrator de Rössler e o atrator de Hénon.

Quatérnios

Os quatérnios são um sistema numérico proposto por Hamilton em 1843. Eles são descritos por Burton (2011) como sendo números hipercomplexos compostos por quatro termos, na forma

()

onde *a*, *b*, *c* e *d* são números reais e *i*, *j* e *k* são imaginários, de forma que . Hamilton acreditava que os quatérnios seriam a ferramenta correta para fornecer uma descrição matemática do tempo e espaço, visto que três coordenadas são necessárias para representação do espaço e uma para o tempo, do mesmo modo que ocorre em um quatérnio.

O uso deste sistema numérico permite ampliar os estudos sobre sistemas dinâmicos. Hart, Sandin e Kauffman (1989) mencionam que a dinâmica que funções quadráticas ocorre também em espaços 4D, e descrevem a utilização de quatérnios para geração de conjuntos de Julia de dimensões maiores que 2D.

Visto que o plano complexo C é um subconjunto dos quatérnios, os mesmos conjuntos de Julia que existem no plano complexo existem também nos quatérnios, porém muitas vezes com extensões fora de C. Os autores (1989) comentam que uma propriedade interessante da geração de conjuntos de Julia com quatérnios é que os mesmos conjuntos podem apresentar formas completamente diferentes apenas alterando sua rotação sobre a origem, visto que as intersecções entre as partes imaginárias são alteradas, o que não acontece no plano complexo.

A geração de fractais com quatérnios pode ser muito custosa em termos computacionais. Isto se deve ao fato de que, como os quatérnios são números com quatro dimensões, é necessária uma representação 3D de um espaço 4D para que se possa visualizar o fractal. Além disso, os cálculos com quatérnios são naturalmente mais difíceis do que com números complexos pelo fato de os números possuírem duas partes a mais.

Crane (2005) disponibiliza em sua página um projeto para geração de fractais com quatérnios baseados em GPU, utilizando programação para *pixel* e *vertex shaders* através de Cg (C for Graphics). A é um exemplo de um conjunto de Julia gerado a partir de quatérnios:

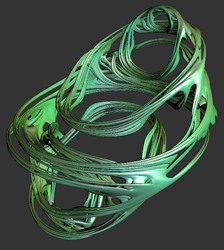


Figura – Representação 3D de um conjunto de Julia gerado com quatérnios.

Fonte: Crane (2005).

Esta seção encerra a fundamentação teórica sobre fractais. O próximo capítulo abordará conceitos de programação paralela, de modo que, juntamente com os conceitos descritos neste capítulo, possa prover embasamento teórico necessário para o desenvolvimento dos algoritmos utilizados para os testes deste estudo.

# PROGRAMAÇÃO PARALELA

A ideia de executar múltiplas tarefas de forma simultânea em uma aplicação não é algo novo. Conforme Amorim, Barbosa e Fernandes (1988, p. 1), “a ideia de incluir paralelismo nos sistemas de computação é tão antiga quanto os computadores eletrônicos. Trabalhos desenvolvidos por von Neumann na década de 40 já discutiam a possibilidade de algoritmos paralelos para a solução de equações diferenciais”. Dongarra et al. (2003) afirmam que a computação paralela não é somente uma estratégia para obter alto desempenho, mas também uma visão de como a computação pode escalar de um único processador até um poder computacional virtualmente ilimitado.

Wilkinson e Allen (2005) explicam que existem dois tipos básicos de computadores paralelos: os multiprocessadores com memória compartilhada, e os multicomputadores com memória distribuída. O primeiro tipo caracteriza-se por ser um modelo onde existe um espaço único de endereçamento de memória visível a todos os processadores. Desta maneira, o código executável deve ser alocado nesta memória compartilhada para que cada processador o execute.

O segundo tipo consiste em um *cluster* de computadores conectados por uma rede. Cada computador possui processador e memória, porém, esta memória não é acessível pelos outros computadores. Desta forma, os computadores precisam trocar mensagens com os dados através da rede.

Os autores (2005) citam que o paradigma baseado em trocas de mensagens geralmente não é tão atrativo aos desenvolvedores quanto o paradigma baseado em memória compartilhada porque tais trocas são mais propensas a erros e tornam os programas difíceis de serem depurados. Assim, surgiu o conceito de um sistema distribuído de memória compartilhada. A ideia é que a memória seja distribuída fisicamente junto com cada processador, porém cada processador tem acesso à memória inteira utilizando um espaço de endereçamento único. O acesso a um endereço de memória que não seja local é feito utilizando troca de mensagens.

Para o desenvolvimento de aplicações utilizando estes modelos de processamento, algumas linguagens, APIs e extensões foram criadas, com o propósito de abstrair os detalhes das arquiteturas de cada plataforma. Wilkinson e Allen (2005, p. 247) mencionam que, no entanto, destas linguagens de programação, nenhuma até então foi universalmente aceita, e citam exemplos como Compositional C++, Fortran M e High Performance Fortran (HPF). Em contrapartida, o uso de APIs para definir paralelismo em conjunto com linguagens de programação já consolidadas é uma abordagem mais atrativa aos desenvolvedores. Dentre as diversas APIs disponíveis atualmente, destacam-se MPI, baseada no modelo de troca de mensagens, e OpenMP, baseado no modelo de memória compartilhada. Os dois modelos são amplamente aceitos atualmente.

Este capítulo é destinado a apresentar estas duas APIs utilizadas para programação em sistemas paralelos. Um maior enfoque será dado à API OpenMP, que foi a escolhida para o desenvolvimento deste estudo. Ao final do capítulo, traz-se uma discussão onde são comparadas as duas abordagens, de modo a justificar a escolha pelo modelo OpenMP.

MPI

Conforme Dongarra et al. (2003, p. 314), “a abordagem baseada em troca de mensagens é, de longe, a mais amplamente utilizada para computação paralela, ao menos em grandes sistemas”. Esta abordagem, segundo os autores (2003), apresenta duas grandes vantagens: a primeira é a portabilidade, de modo que virtualmente qualquer conjunto de computadores pode ser utilizado para executar uma aplicação paralela que utilize troca de mensagens; e a segunda é o fato de proporcionar aos desenvolvedores controle explícito sobre a memória.

Embora este modelo de programação seja altamente portável pelo fato de poder ser facilmente implementado em qualquer plataforma, uma aplicação é portável apenas se o sistema para o qual foi desenvolvida for amplamente utilizado. Contudo, esta não era a realidade até o final da década de 80. Segundo Quinn (2003), àquela época o ambiente de desenvolvimento para computadores paralelos era geralmente baseado em uma linguagem sequencial, como C ou Fortran, com extensões através de bibliotecas que permitiam que os processos trocassem mensagens. Cada fabricante possuía um conjunto próprio de funções, de modo que um programa compilado para um tipo de computador não poderia executar em outro.

Tendo em mente, assim, que um único padrão bem especificado traria benefícios aos desenvolvedores e fabricantes, foi criada a primeira versão do MPI, o padrão atualmente mais difundido para a programação baseada em troca de mensagens (DONGARRA ET AL., 2003). O MPI (do inglês *message passing interface*) é uma especificação de uma biblioteca inicialmente projetada por um amplo grupo composto por fabricantes de computadores paralelos, pesquisadores e desenvolvedores, com o intuito de prover um padrão portável, eficiente e escalável (SNIR ET AL., 1996). Estas características são alcançadas pelo fato de o padrão MPI utilizar conceitos e boas práticas de diversas outras bibliotecas para processamento com troca de mensagens, resultando, assim, numa especificação que inclui uma grande variedade de funções, as quais podem ser utilizadas no desenvolvimento de aplicações complexas. A biblioteca provê suporte às linguagens C e Fortran (DONGARRA ET AL., 2003).

A primeira versão da especificação foi disponibilizada ao público em junho de 1994, e a versão 1.1 no ano seguinte (SNIR ET AL, 1996). Atualmente, a versão mais atual do padrão é o MPI-2, a qual disponibiliza diversas melhorias e adições à versão 1.1.

Dongarra et al. (2003) afirmam que, apesar que o padrão MPI ser um sistema complexo, apenas seis de suas muitas funções são necessárias para resolver uma ampla gama de problemas. Estas funções são as seguintes:

* MPI\_INIT: inicia uma computação com MPI;
* MPI\_FINALIZE: encerra a computação;
* MPI\_COMM\_SIZE: para determinar o número de processos;
* MPI\_COMM\_RANK: para determinar o identificador de um processo;
* MPI\_SEND: envia uma mensagem; e
* MPI\_RECV: recebe uma mensagem.

OPENMP

Dentre as diversas opções disponíveis para programação paralela em sistemas com memória compartilhada, uma das mais populares é o modelo OpenMP. O OpenMP é um padrão desenvolvido no final dos anos 90 que consiste em um conjunto de diretivas de compilação que descrevem o paralelismo no código-fonte, juntamente com uma biblioteca de rotinas e algumas variáveis de ambiente. Estas diretivas, bibliotecas e variáveis de ambiente, em conjunto, formam a API OpenMP, disponível para Fortran e C/C++, sendo utilizadas através de comentários em Fortran, e através de *pragmas* em C/C++ (CHANDRA ET AL., 2001). Dongarra et al. (2003) ainda definem que o OpenMP é uma interface *multithreading* projetada especificamente para o suporte a programas paralelos de alta performance.

A opção de utilizar diretivas de compilação ao invés de basear toda a implementação em extensões de linguagem tem como principal atrativo a possibilidade de utilizar um programa inicialmente desenvolvido puramente de forma sequencial e especificar o paralelismo onde necessário. Desta forma, um compilador normal simplesmente ignorará as diretivas OpenMP e gerará um executável sequencial, enquanto um compilador OpenMP produzirá uma versão paralela do mesmo programa (WILKINSON E ALLEN, 2005). Chandra et al. (2001) mencionam ainda que, inicialmente, foi considerada uma abordagem totalmente baseada em bibliotecas, mas que esta opção foi rejeitada por dois fatores: o fato de um compilador que não suporta OpenMP ignorar automaticamente as diretivas que especificam o paralelismo torna o código muito mais fácil de ser portado; e a possibilidade de realizar otimizações baseadas no compilador com o uso de tais diretivas. Visto que uma abordagem completamente voltada a diretivas também tornaria o desenvolvimento difícil, optou-se por utilizar algumas rotinas em conjunto com as mesmas.

A API OpenMP utiliza o modelo de execução paralela *fork-join*. Wilkinson e Allen (2005, p. 233) conceituam este modelo, onde

um comando *fork* gera um novo caminho para um processo concorrente e estes processos concorrentes usam um comando *join* ao final das execuções. Quando todos os comandos *join* são alcançados, a execução continua de modo sequencial. Para mais processos concorrentes são necessários comandos *fork* adicionais.

A especificação da API, definida pelo OpenMP Architecture Review Board (2011), define que a execução de um programa OpenMP começa com uma única *thread*, chamada de *thread* mestra, a qual executa sequencialmente. A diretiva *parallel* cria um grupo de *threads* e especifica um bloco que será executado por este grupo de *threads* em paralelo.

Chandra et al. (2001) dividem as rotinas e diretivas do modelo OpenMP em três categorias: estruturas de controle paralelo, ambiente de dados e sincronização.

A primeira categoria – a mais relevante para este trabalho – consiste em estruturas que alteram o fluxo de controle em um programa. A diretiva básica é a já citada *parallel*. As outras diretivas nesta categoria são utilizadas para dividir o trabalho entre um grupo de *threads* em paralelo. Dentre estas, destacam-se as diretivas para definição de paralelismo em laços (*loops)*, como *for* e *do*, nas quais focar-se-á para os propósitos deste trabalho. O padrão OpenMP provê diretivas específicas para tais situações, visto que frequentemente o paralelismo pode ser encontrado em laços. A Figura 11 ilustra a execução de um programa que faça uso da diretiva *do* para especificar as tarefas a serem executadas em paralelo:

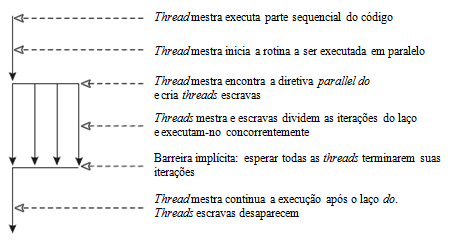


Figura – Exemplo de execução paralela de um laço do tipo *do*.

Fonte: Adaptado de Chandra et al. (2001, p. 25).

Juntamente com estas diretivas, pode-se fazer uso de alguns qualificadores para especificar o modo como as iterações serão distribuídas entre as *threads* (DONGARRA ET AL., 2003). Estes modos podem ser os seguintes, como descrito por Barney (2012):

* *Static*: as iterações são divididas em blocos de maneira contígua para execução pelas *threads*. O número de iterações por *thread* é definido pelo parâmetro adicional *chunksize* (tamanho do bloco). Quando este parâmetro não é especificado, as iterações são divididas em *chunks* de tamanho aproximadamente igual;
* *Dynamic*: este método distribui as iterações pelas *threads* à medida que estas finalizam suas tarefas. Quando uma *thread* encerra um bloco (*chunk*), ela dinamicamente recebe outro para processar. O tamanho padrão de *chunk* é igual a 1;
* *Guided*: similar ao modo *dynamic*, porém, aqui o tamanho dos blocos diminui conforme as iterações são processadas;
* *Runtime*: a distribuição das iterações é feita em tempo de execução, e não em tempo de compilação;
* *Auto*: a distribuição é delegada ao compilador e sistema de execução.

A segunda categoria de extensões da linguagem que compõem o padrão OpenMP, a qual abrange as diretivas de ambiente de dados, é composta pelas diretivas para definição do escopo das variáveis processadas pelas *threads*. Para tanto, três atributos básicos podem ser utilizados, conforme Chandra et al. (2001):

* *Shared*: uma variável com este atributo possui um único espaço em memória para a duração inteira de uma seção paralela. Desta forma, todas as *threads* que referenciem esta variável acessarão o mesmo endereço de memória, dado que este é compartilhado pelas *threads*;
* *Private*: a definição de escopo privado faz com que a variável tenha múltiplas posições em memória, uma para o contexto de execução de cada *thread*. A posição de memória para uma *thread* é inacessível às outras.
* *Reduction*: é um atributo especial utilizado para realizar operações de redução aritmética.

Por fim, a terceira categoria engloba as diretivas de sincronização entre *threads*. A comunicação entre elas é feita através de variáveis compartilhadas, e, portanto, frequentemente é necessário que haja algum tipo de coordenação nos acessos a tais variáveis. Caso contrário, pode ocorrer que múltiplas *threads* tentem modificar uma mesma variável ao mesmo tempo, ou ler um valor que esteja sendo modificado por outra *thread*.

Para prover a sincronização entre as *threads*, OpenMP faz uso de duas formas básicas: exclusão mútua, expressa pela diretiva *critical*, na qual uma *thread* possui acesso exclusivo à variável compartilhada para uma determinada seção de código; e sincronização de eventos, por meio da diretiva *barrier*, de modo que, uma vez alcançada esta diretiva, o programa aguarda todas as *threads* chegarem até a mesma, para então continuarem a execução. A diretiva *parallel* possui uma barreira de sincronização implícita no final do bloco, como previamente descrito na Figura 11.

DISCUSSÃO

Chapman, Jost e van der Pas (2008) defendem que MPI provê um excelente nível de portabilidade, sendo amplamente utilizado em computação de alto desempenho, onde os problemas são tão grandes que são necessários diversos computadores para resolvê-los. Entretanto, tal portabilidade vem com o custo de que a criação de programas baseados nesta API geralmente requer extensas modificações ao código sequencial original, o que torna o desenvolvimento mais complexo.

Chandra et al. (2001) compartilham desta opinião, e mencionam ainda que a programação utilizando MPI é mais complexa porque a utilização de um modelo baseado em troca de mensagens requer que as estruturas de dados do programa sejam explicitamente particionadas, o que faz com que as aplicações frequentemente necessitem ser completamente paralelizadas.

Dongarra et al. (2003) resumem afirmando que o uso de MPI é particularmente adequado em sistemas onde a portabilidade, tanto no espaço (entre diferentes sistemas que existem no momento) quanto no tempo (entre diferentes gerações de computadores) é um fator importante. Em contrapartida, os mesmos autores (2003) recomendam o uso de OpenMP quando o objetivo é alcançar um nível modesto de paralelismo em um computador com memória compartilhada. Neste caso, a simplicidade do modelo OpenMP caracteriza uma vantagem significativa. Os autores (2003) comentam ainda que o modelo OpenMP é frequentemente utilizado em conjunto com MPI no caso de sistemas com memória distribuída, de modo a explorar as vantagens das duas técnicas, com OpenMP usado para processamento em cada nodo e MPI para efetuar a troca de mensagens entre estes.

Para o presente trabalho, a escolha pelo uso da API OpenMP teve dois fatores como principal motivação:

* o ambiente de desenvolvimento utilizado durante este estudo, caracterizado por ser um sistema *multi-core* com memória compartilhada; e
* o potencial uso dos algoritmos desenvolvidos com modelo sequencial como base para a implementação paralela, através da adição de diretivas de compilação. Com isto, pode-se aproveitar quase totalmente o código desenvolvido para os testes dos modelos sequenciais, mudando apenas os parâmetros de compilação para indicar se o programa deve ser compilado com OpenMP ou não, permitindo, assim, que versões paralelas e sequenciais de um mesmo programa sejam facilmente geradas.

O uso de MPI, entretanto, pode ser abordado em um estudo futuro, podendo-se comparar os resultados de sua utilização com os obtidos neste trabalho.

Esta discussão encerra este capítulo, no qual a ênfase foi em modelos de programação paralela tradicionais baseados em CPUs. Indo de encontro ao objetivo deste trabalho de avaliar o desempenho de algoritmos para a geração de fractais com base em uma arquitetura *many-core,* o próximo capítulo é destinado a apresentar conceitos, arquiteturas e modelos de programação baseados em GPU.

# PROGRAMAÇÃO COM GPUS

A utilização de GPUs como ferramentas de computação para aplicações outras que não gráficas é motivada especialmente pelo grande número de unidades lógicas e aritméticas (ULA) presentes em tais dispositivos.

Kirk e Hwu (2010, p. 5) afirmam que “a filosofia de projeto de GPUs é modelada pelo rápido crescimento da indústria de *video games*, a qual exerce uma enorme pressão para que seja possível desempenhar um número massivo de operações de ponto-flutuante por *frames* de vídeo nos jogos modernos”. Os mesmos autores (2010) ainda comentam que, deste modo, os fabricantes são forçados a encontrar soluções que sejam capazes de maximizar o número de tais operações. Até o momento, a principal solução para esta demanda consiste em otimizar a execução para que se tenha um *throughput* de grandes números de *threads*. Como resultado, uma área maior dos chips das GPUs é dedicada a cálculos de ponto-flutuante. A Figura 12 ilustra esse conceito, mostrando que as GPUs possuem, deste modo, áreas de controle e memória reduzidas, porém com um número muito maior de ULAs:

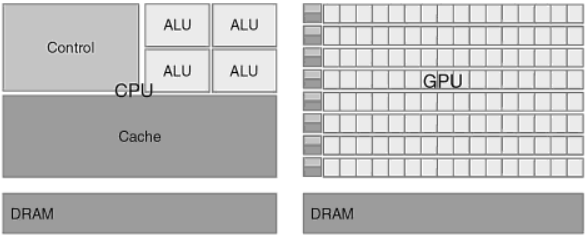


Figura – Comparativo entre os projetos de uma CPU e de uma GPU.

Fonte: Kirk e Hwu (2010, p. 4).

Assim, à medida que o *hardware* das GPUs evolui e estas possuem cada vez mais processadores unificados, tais dispositivos cada vez mais adquirem características de computadores paralelos de alto desempenho.

Este fato fez com que pesquisadores começassem a explorar o potencial das GPUs para resolver problemas científicos e de engenharia de alta demanda computacional. Contudo, como os chips gráficos eram projetados apenas para terem compatibilidade com as interfaces de programação de aplicações (*application programming interface* – API) gráficas, era necessário transformar o problema a ser resolvido em operações gráficas para que se conseguisse acessar os recursos computacionais da GPU. Esta técnica ficou conhecida como GPGPU, acrônimo para unidade de processamento gráfico de propósito geral (*general-purpose graphics processing unit*) (KIRK e HWU, 2010). Tais restrições mostraram que o modelo de programação com GPGPUs não era adequado para aplicações numéricas, o que, consequentemente fez com que, até 2006, os chips gráficos não fossem amplamente utilizados como ferramentas computacionais de propósito geral.

A ideia de utilizar os dispositivos gráficos como ferramentas de computação de propósito geral não foi abandonada, entretanto, e, para suprir as limitações inerentes ao modelo de programação com APIs, alguns outros modelos foram surgindo, de modo a facilitar o desenvolvimento de aplicações.

A empresa NVIDIA, enquanto desenvolvia sua arquitetura de GPUs Tesla, percebeu que esta teria um potencial de uso ainda maior se a GPU pudesse ser vista como um processador, com um modelo de programação no qual pudessem ser declarados explicitamente os aspectos paralelos da aplicação (KIRK e HWU, 2010). Assim, em 2006 a empresa apresentou CUDA, uma arquitetura paralela de propósito geral com um novo modelo de programação e conjunto de instruções que utiliza o motor gráfico de suas GPUs para resolver problemas computacionais complexos (NVIDIA, 2010a).

O modelo de programação para CUDA, todavia, permite que as aplicações sejam desenvolvidas para executarem apenas com as placas gráficas da NVIDIA. Com isto, outros modelos de programação com GPUs foram surgindo nos anos seguintes, oferecendo compatibilidade com outras placas gráficas.

Assim, este capítulo é destinado à apresentação dos principais modelos de programação utilizando GPUs como ferramenta de computação de propósito geral. Inicialmente apresenta-se o modelo de programação heterogêneo OpenCL e as arquiteturas Intel Larrabee e AMD Accelerated Parallel Processing, opções alternativas à abordada neste trabalho. Com isto, pretende-se trazer a discussão sobre e justificar a escolha utilizada para o desenvolvimento deste trabalho. O capítulo 4 abordará em mais detalhes a arquitetura CUDA, escolhida para as implementações deste estudo.

OPENCL

A evolução das CPUs e GPUs e a utilização destes processadores em sistemas computacionais paralelos trazem a necessidade de prover um modelo de programação que possa ser adequadamente utilizado para o desenvolvimento em plataformas heterogêneas. O modelo de programação OpenCL (*Open Computing Language*) surge para suprir esta necessidade. Desenvolvido pelo Grupo Khronos, é um padrão aberto para programação paralela de propósito geral para CPUs, GPUs ou outros processadores. Desta forma, os desenvolvedores têm acesso a plataformas heterogêneas de modo portátil e eficiente (KHRONOS GROUP, 2011).

O Grupo (2011) ainda define o padrão OpenCL como um *framework* que inclui uma API para coordenação de processamento paralelo entre processadores heterogêneos, uma linguagem de programação e um sistema *runtime* para o desenvolvimento de aplicações. O *framework*, entre outras características, suporta modelos de programação paralela baseados em paralelismo de dados ou de tarefas, e interopera de maneira eficiente com outras APIs gráficas, como OpenGL. O modelo de processamento é ilustrado pela Figura 13, onde múltiplos dispositivos (*devices*) OpenCL são conectados a um hospedeiro (*host*). Os dispositivos são divididos em unidades computacionais (*compute units*), as quais, por sua vez, são divididas em elementos de processamento (*processing elements*).



Figura – Modelo de processamento OpenCL

Fonte: Khronos Group (2011, p. 22).

A execução de um programa OpenCL ocorre em duas partes: os *kernels* são executados nos dispositivos e o programa *host* é executado no hospedeiro. A aplicação envia comandos do hospedeiro para executar tarefas nos elementos de processamento de um dispositivo (KHRONOS GROUP, 2011).

AMD ACCELERATED PARALLEL PROCESSING (ATI STREAM)

Accelerated Parallel Processing (APP), antigamente conhecida como Stream, é a solução da empresa AMD/ATI para computação paralela de propósito geral com GPUs. Esta solução utiliza um modelo de programação baseado em OpenCL, descrito no capítulo anterior.

Um sistema AMD APP é composto por GPUs AMD e uma pilha de *software* (*software stack*). Esta pilha é composta por bibliotecas, pelo ambiente de execução OpenCL e por uma camada de abstração denominada CAL – *Compute Abstraction Layer*. Esta camada desempenha um importante papel no ambiente APP, provendo abstrações sobre o *hardware* das GPUs para a camada de software. CAL também proporciona controle sobre os dispositivos, controle de recursos, serviços de geração de código e carregamento e execução de *kernels* (AMD, 2010). A Figura 14 mostra como ocorre a integração entre os componentes de um sistema AMD APP:

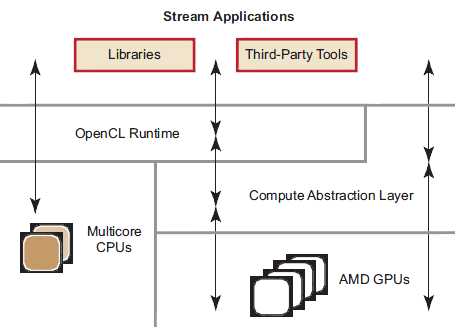


Figura – Relação entre os componentes em um ambiente AMD APP

Fonte: AMD (2011, p. 18).

Os núcleos de processamento são chamados *stream cores* nesta arquitetura, os quais são responsáveis por executar os *kernels*, cada um operando sobre um fluxo (*stream*) independente de dados. Cada instância de um *kernel* executando em uma unidade computacional é denominada um item de trabalho (*work-item*). A GPU então organiza os itens de trabalho em um grupo de *stream cores* até que todos os itens tenham sido processados (AMD, 2011).

CAL interage com os *stream processors* – nome dado aos processadores do dispositivo que executam os *kernels* – como estes sendo uma matriz de processadores SIMD (*Single Instruction, Multiple Data*), cada um operando independente e paralelamente sobre os fluxos de dados.

INTEL LARRABEE

Larrabee é o codinome da arquitetura *many-core* apresentada pela empresa Intel, baseada em uma versão estendida do conjunto de instruções x86, para computação gráfica e de alto desempenho. Glaskowsky (2008) define a arquitetura como sendo basicamente um modelo que utiliza múltiplos núcleos x86, como os de uma CPU, para implementar um processador gráfico.

A principal motivação por trás deste conceito é prover uma solução mais flexível que as GPUs existentes. Para Seiler et al. (2008), a programabilidade de GPUs para computações de propósito geral é restrita devido a limitações no modelo de memória e pelos blocos de funções fixas que controlam as *threads* executando paralelamente.

Deste modo, esta flexibilidade, conforme Seiler et al. (2008), é alcançada na arquitetura Larrabee por meio de suporte a sub-rotinas e *page faults* (faltas de página), além do fato de que algumas operações que tradicionalmente são realizadas pelas GPUs são executadas inteiramente em *software* com Larrabee, como rasterização de imagens e *post-shader blending*. Glaskowsky (2008), entretanto, não vê este fato como sendo um ponto a favor da arquitetura, afirmando que uma solução baseada em *software* certamente consumirá mais recursos computacionais.

Seiler et al. (2008) ainda definem a arquitetura como sendo derivada dos processadores Pentium, com seus processadores executando em ordem. Os núcleos de processamento foram aumentados com unidades de processamento vetorial. Cada núcleo possui apenas duas unidades de execução: uma para instruções escalares e outra para vetoriais. Em comparação com CPUs *quad-core*, com mais de seis unidades de execução, esta é uma redução significativa de complexidade, o que torna a arquitetura adequada para processamento linear e com código previsível (GLASKOWSKY, 2008). Além disso, cada núcleo acessa subconjuntos próprios de 256kB de *cache* de nível L2 coerente. A *cache* L1 possui 32kB para instruções e mais 32kB para dados. A Figura 15 mostra os blocos de execução de um núcleo Larrabee:



Figura – Núcleo de um processador Larrabee

Fonte: Seiler et al. (2008, p. 3).

O modelo de programação para a arquitetura da Intel se chama Larrabee Native. Segundo Seiler et al. (2008), este modelo assemelha-se ao já utilizado para programação para arquiteturas *multi-core* x86. Ainda, Larrabee Native proporciona um compilador C/C++ que compila os programas para o conjunto de instruções x86 utilizado pela arquitetura. Desta forma, muitas das aplicações escritas nessas linguagens podem ser recompiladas para o uso com Larrabee sem necessitarem de nenhuma modificação, o que por si só já traz ganhos enormes de produtividade para os desenvolvedores (SEILER et al., 2008).

O projeto, entretanto, foi cancelado em 2010. A empresa anunciou que não traria um produto discreto para gráficos ao mercado em curto prazo (SMITH, 2010). Apesar disso, algumas ideias foram reaproveitadas em outros desenvolvimentos, como a arquitetura MIC e a família de processadores Knight’s Ferry.

Esta seção finaliza a apresentação dos principais modelos de programação com GPUs. O próximo capítulo é destinado à apresentação do modelo de programação e arquitetura CUDA, escolhido para o desenvolvimento deste trabalho.

# A Arquitetura CUDA

Conforme já mencionado no capítulo 3, a NVIDIA apresentou, em novembro de 2006, sua arquitetura paralela de propósito geral, chamada CUDA – acrônimo para *Compute Unified Device Architecture*. A arquitetura foi inicialmente utilizada na placa gráfica GeForce 8800 GTX, a primeira GPU com suporte a DirectX 10 e contendo uma série de novos componentes projetados especificamente para computação com GPU (SANDERS E KANDROT, 2011).

Visto que a empresa pretendia que seus dispositivos gráficos pudessem ser utilizados para computação de propósito geral, os processadores foram projetados em conformidade com as especificações da IEEE referentes a aritmética de ponto flutuante de precisão simples e contêm um conjunto de instruções voltado a computação genérica, e não apenas para processamento gráfico. Além disso, para simplificar o modelo de programação e possibilitar que um maior número de desenvolvedores utilizasse a arquitetura, a NVIDIA introduziu uma extensão da linguagem C, chamada CUDA C, contendo algumas instruções especiais para prover o acesso aos dispositivos (SANDERS E KANDROT, 2011). Deste modo, mantendo um conjunto mínimo de extensões à linguagem C, obtém-se uma baixa curva de aprendizado para o desenvolvimento de aplicações com CUDA. O ambiente de programação ainda provê a possibilidade de utilizar uma extensão da linguagem FORTRAN ou o previamente descrito modelo OpenCL (NVIDIA, 2010a).

Kirk e Hwu (2010) descrevem o modelo de programação para a arquitetura CUDA como sendo do tipo SPMD (*single-program, multiple-data*). Os autores explicam que este modelo é diferente do tipo SIMD (*single-instruction, multiple data*), pois em um ambiente SPMD as unidades de processamento executam o mesmo programa em partes diferentes dos dados, mas não necessitam obrigatoriamente estar executando a mesma instrução ao mesmo tempo, como ocorre em sistemas do tipo SIMD. A NVIDIA (2010a), entretanto, cita sua arquitetura como sendo do tipo SIMT (*single-instruction, multiple-thread*), mencionando que, apesar de ambos os modelos caracterizarem-se pelo fato de que uma única instrução controla múltiplos elementos de dados, as instruções em uma arquitetura SIMT especificam o comportamento da execução e das ramificações de uma *thread* sozinha.

Farber (2011) complementa ainda dizendo que GPGPUs não são verdadeiramente do tipo SIMD porque são compostas de vários multiprocessadores, sendo que cada um pode estar executando diferentes instruções de um *kernel* em um dado momento.

Aparentemente não existe um consenso quanto à terminologia mais adequada disponível na literatura. Para efeito de simplificação e entendimento, a arquitetura será referenciada neste trabalho como sendo do tipo SIMD.

As próximas seções detalham os principais aspectos do modelo de programação CUDA, com foco na organização de *threads* e nos diferentes níveis de memória.

*KERNELS* E HIERARQUIA DE *THREADS*

Um dos principais conceitos presentes no modelo de programação para CUDA é o de *kernels*, os quais, quando invocados, são executados *n* vezes em paralelo por *n* *threads* diferentes (NVIDIA, 2010a). As *threads*, por sua vez, são organizadas em blocos (*blocks*) tridimensionais, e tais blocos estão contidos, por sua vez, em grades (*grids*) bidimensionais. Esta organização, conforme a empresa (2010a), permite que o processamento em vetores, matrizes e volumes seja feito de maneira natural. O número de *threads* por bloco é limitado, visto que, por executarem em um mesmo núcleo de processamento, compartilham da mesma memória disponível para este núcleo. A empresa (2010a) ainda menciona que em GPUs atuais os blocos podem conter até 512 *threads*.

A hierarquia de *threads* é ilustrada na Figura 16:

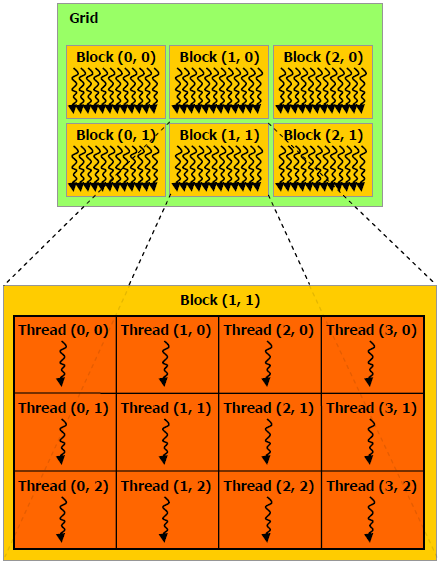


Figura – A hierarquia de *threads* na arquitetura CUDA

Fonte: NVIDIA (2010a, p. 9).

Conforme já citado, os *kernels* são rotinas que são executadas paralelamente por diversas *threads* nos núcleos da GPU. A definição de um *kernel* é feita simplesmente com a adição da palavra-chave *\_\_global\_\_* antes do nome da rotina. Além desta, duas outras palavras-chave são utilizadas em CUDA C de modo a definir onde uma rotina deve ser executada. Um resumo destas palavras-chave é exibido no quadro abaixo:



Quadro – Palavras-chave para definição de um *kernel*.

Fonte: Adaptado de Kirk e Hwu (2010, p. 52).

Visto que todas as *threads* de um *kernel* executam a mesma porção de código, faz-se necessário um modo de distingui-las, de modo que cada uma acesse e processe as porções de dados às quais cada uma está destinada. Para tanto, CUDA provê variáveis que identificam os índices e das *threads* e blocos. Esta identificação é feita usando as variáveis *blockIdx* (para o índice do bloco) e *threadIdx* (para o índice da *thread*). Além disso, são disponibilizadas as variáveis *gridDim* e *blockDim* com os valores da dimensão da grade e de cada bloco, respectivamente (KIRK E HWU, 2010).

Ainda conforme os autores (2010), ao iniciar-se um *kernel* deve-se informar os parâmetros referentes à configuração de execução deste. O primeiro parâmetro se refere à dimensão da grade em termos de número de blocos, enquanto o segundo parâmetro informa as dimensões de cada bloco em termos de número de *threads*. Cada um destes parâmetros é do tipo *dim3*, o qual é uma estrutura composta por três campos do tipo inteiro sem sinal: *x*, *y* e *z*. Visto que grades possuem uma estrutura bidimensional, o terceiro parâmetro é ignorado ao se especificar o tamanho da mesma.

A figura a seguir ilustra um exemplo de configuração de um *kernel*, composto por uma grade de 4 blocos e cada bloco contendo 16 *threads*:



Figura – Exemplo de chamada de um *kernel*.

Fonte: Kirk e Hwu (2010 p. 62).

*WARPS*

Quando um *kernel* é chamado, o multiprocessador cria, gerencia, agenda e executa as *threads* em grupos, geralmente compostos de 32, chamados *warps*. Farber (2011) cita que o *warp* é a unidade básica de escalonamento de processos dentro de um SM. Esta organização é necessária para reduzir os efeitos de algumas limitações de *hardware*, especialmente para otimizar os acessos à memória.

A estratégia de agrupar *threads* em *warps* permite que os processadores em um dispositivo CUDA executem operações de longa latência de forma eficiente, quando há um número adequado de *warps*. Isto se deve ao fato de que CUDA possui um mecanismo de escalonamento que “preenche” as latências de operações muito custosas com operações de outros *warps*. Esta é a principal razão pela qual as GPUs não possuem tanta área de seus chips dedicadas a memória cache ou mecanismos de predição de desvios como as CPUs, o que permite que as GPUs dediquem mais área a suas unidades de cálculo de ponto-flutuante (KIRK E HWU, 2010).

Dado que um *warp* executa uma instrução por vez, a eficiência ótima é alcançada apenas quando todas as *threads* de um *warp* percorrem o mesmo caminho de execução. Ou seja, caso haja divergência na execução das *threads,* por exemplo, devido a desvios condicionais, o *warp* é executado de maneira sequencial, percorrendo cada caminho do desvio, para que apenas ao final as *threads* convirjam novamente ao mesmo caminho de execução (NVIDIA, 2010a). Projetar a aplicação de forma a tentar evitar ao máximo divergências dentro dos *warps* caracteriza um grande desafio, mesmo que muitas vezes não há como eliminar completamente tais divergências.

MEMÓRIAS

Para fazer uso eficiente do poder computacional da arquitetura CUDA, é necessário utilizar apropriadamente a memória do dispositivo, para que esta não seja um fator limitador de desempenho nas aplicações. Kirk e Hwu (2010) comentam que um dos principais gargalos no desempenho de uma aplicação CUDA é gerado devido ao mau uso desta memória, especialmente a memória global. Tal memória, pelo fato de ser geralmente construída utilizando-se memória dinâmica de acesso aleatório (DRAM), tende a possuir altas latências e largura de banda limitada – de 177 GB/s, conforme Farber (2011). Assim, uma aplicação que faça uso de muitas *threads* acessando a memória global pode facilmente congestionar o acesso a esta, acarretando em perda de desempenho.

Outro fator limitador de desempenho é a taxa de transferência de dados entre o host e o dispositivo. A NVIDIA (2010b) afirma que o pico da largura de banda entre a memória do dispositivo e a GPU é muito mais alto que o pico da largura de banda entre a memória do host e a memória do dispositivo, com valores de 141 GB/s (para a placa GeForce GTX 280, por exemplo) e 8 GB/s, respectivamente. Esta limitação de 8 GB/s se deve ao pico do barramento PCIe x16 Gen2. Isto demonstra a importância de minimizar tais transferências, mantendo os dados o máximo possível na memória do dispositivo.

Para contornar tais problemas, CUDA provê diferentes tipos de memória que podem ser utilizadas para diminuir o tráfego à memória global. Ao alocar uma variável em um destes tipos de memória, define-se a visibilidade, velocidade de acesso, escopo e tempo de vida da mesma. Estas memórias e algumas de suas características são ilustradas no quadro abaixo, e detalhadas na sequência:



Quadro – Características dos diferentes tipos de memória na arquitetura CUDA.

Fonte: Adaptado de Kirk e Hwu (2010, p.80) e Farber (2011, p. 116).

Farber (2011) comenta a respeito dos diferentes tipos de memória da GPU.

A memória global é a maior disponível em um dispositivo CUDA. Entretanto, é também a mais lenta, de modo que a latência de acesso a essa memória pode ser até 600 vezes maior que a de um registrador. Para reduzir os efeitos desta alta latência e minimizar o congestionamento na memória global, o *hardware* da GPU coalesce os acessos de leitura e escrita. Isto significa que uma única operação é utilizada para retornar os dados de posições consecutivas de memória. Dado que a coalescência é feita por *warps*, o melhor desempenho é alcançado quando os dados estão organizados na memória de modo que todas as *threads* em um *warp* acessem posições sequenciais (KIRK E HWU, 2010).

Os registradores compõem a memória mais rápida da GPU, sendo a única com suficiente largura de banda – conforme Farber (2011), aproximadamente 8 TB/s – e latência baixa a ponto de atingir o pico de desempenho. A NVIDIA (2010a) afirma que cada multiprocessador possui um conjunto de registradores de 32 bits que são divididos entre os *warps*. Geralmente, variáveis declaradas dentro de um *kernel* são alocadas nos registradores, com exceção de grandes estruturas ou *arrays* que consumam muito espaço, ou quando o *kernel* utiliza mais registradores do que há disponíveis no multiprocessador. Nestes casos, o compilador decide alocar os dados na memória local. Entretanto, o número de registradores disponíveis é extremamente reduzido, chegando a apenas 16.384 por multiprocessador em dispositivos com capacidade computacional 1.2.

A memória compartilhada pode ser tanto de 16 kB ou 48 kB por multiprocessador, dividida em 32 bancos com largura de 32 bits. Farber (2011) menciona que, ao contrário da documentação inicial da NVIDIA, a memória compartilhada não é tão rápida quanto os registradores. Porém, com largura de banda de aproximadamente 1,6 TB/s, esta memória é quase dez vezes mais rápida que a memória global, o que a torna um excelente recurso para o desenvolvimento das aplicações. Em circunstâncias ideais, 32 *threads* são capazes de acessar a memória compartilhada paralelamente sem perda de desempenho.

A memória constante tem tamanho igual a 64 kB, e sua principal utilização é em situações onde os dados devem ser disponibilizados para leitura a todas as *threads*. Por isto, é comum que tal memória sirva para armazenar informações de entrada para o processamento a ser feito dentro do *kernel* (KIRK E HWU, 2010). Ainda conforme os autores, a memória constante provê acesso de baixa latência e alta largura de banda quando todas as *threads* em um *warp* acessam um mesmo endereço de memória.

Por fim, CUDA provê ainda um tipo adicional de memória, chamada memória de texturas. Conforme a NVIDIA (2010a), a arquitetura fornece um subconjunto do *hardware* de texturas utilizado pela GPU para gráficos para acessar esta memória. Sanders e Kandrot (2011) mencionam que, visto que tal memória possui um cache dentro do chip, em determinadas situações o uso da mesma trará um aumento de desempenho por necessitar menos acessos à memória do dispositivo. Ainda (2011, p. 116), “os caches de texturas são projetados para aplicações gráficas onde os padrões de acesso à memória exibem uma grande quantidade de localidade espacial”. Isto implica que a memória é otimizada para armazenar em cache os endereços de memória em uma perspectiva 2D, ao invés de prover um padrão de acesso sequencial. Farber (2011, p. 121) resume três importantes características deste tipo de memória: “são geralmente utilizadas em visualização; o cache é otimizado para localidade espacial 2D; e contém apenas 8 kB de cache por SM”.

A Figura 18 ilustra a organização dos diferentes níveis de memória em uma placa gráfica CUDA:

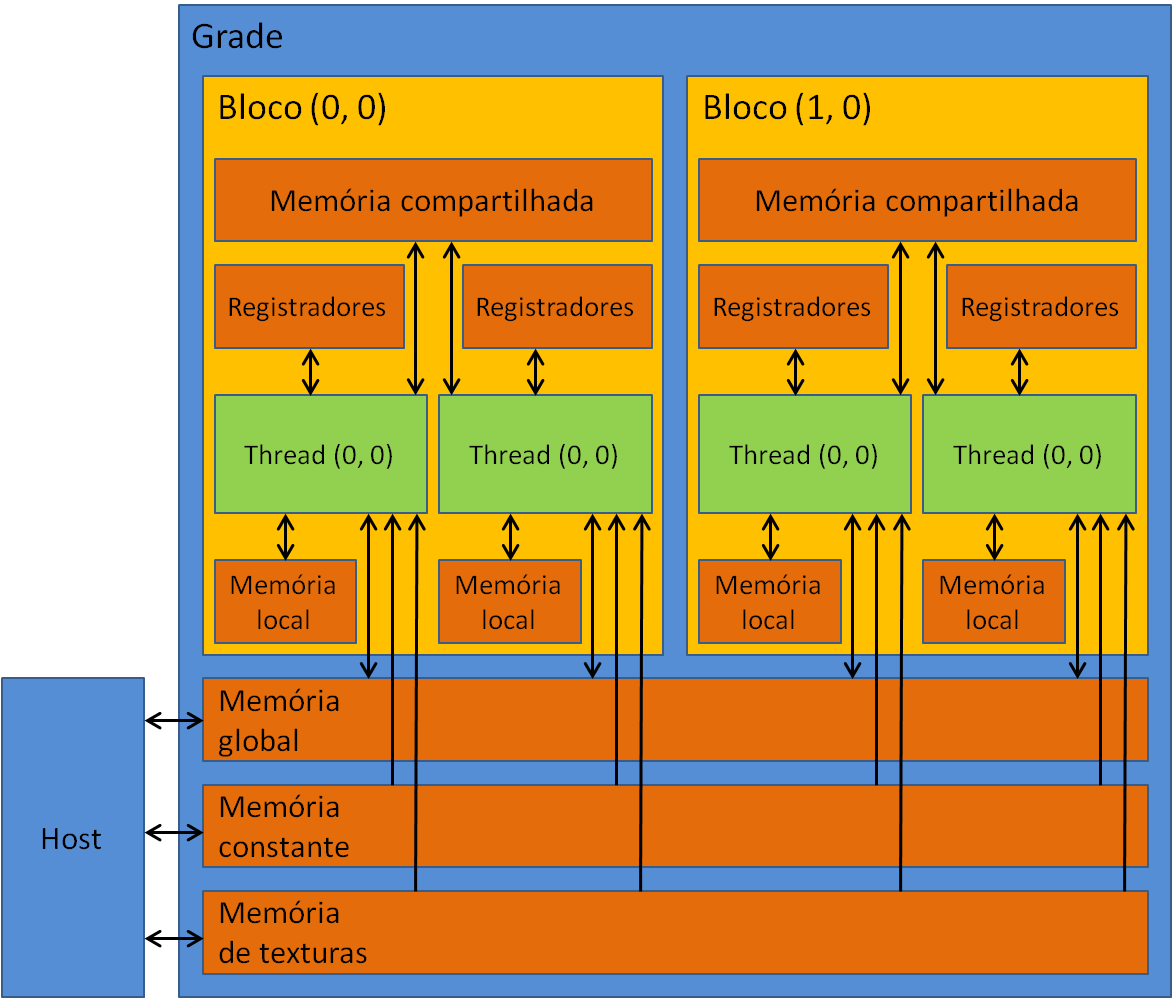


Figura – Modelo de memória de um dispositivo CUDA.

Fonte: Adaptado de Kirk e Hwu (2010, p. 79).

CAPACIDADE COMPUTACIONAL

A NVIDIA categoriza seus dispositivos com base em uma classificação chamada capacidade computacional (do inglês, *compute capability*). Sanders e Kandrot (2011, p. 164) fazem um paralelo desta classificação com as características de CPUs, afirmando que “do mesmo modo que diferentes modelos de CPUs possuem capacidades e conjuntos de instruções variáveis (por exemplo, MMX, SSE, ou SSE2), o mesmo ocorre com os dispositivos na arquitetura CUDA”.

A empresa define a capacidade computacional de seus dispositivos com dois números de revisão, um maior e outro menor. Dispositivos com o mesmo número de revisão maior são baseados na mesma arquitetura, como, por exemplo, as placas baseadas na arquitetura Fermi, onde o número de revisão maior é 2. O número de revisão menor caracteriza melhorias incrementais à arquitetura (NVIDIA, 2010a).

O fato de as melhorias caracterizadas pelo número de revisão menor serem incrementais faz com que novas revisões ainda suportem as características dos modelos anteriores. Deste modo, um dispositivo de capacidade computacional 1.2 suporta todas as características das capacidades computacionais 1.0 e 1.1. A partir do momento que se tem as informações sobre as características presentes em uma determinada capacidade computacional, é possível desenvolver a aplicação e compilá-la de modo a informar que o código só poderá ser executado em dispositivos com uma capacidade computacional mínima (SANDERS E KANDROT, 2011). As características mencionadas até o momento neste trabalho se referem a dispositivos de capacidade computacional 1.x.

Encerra-se aqui a apresentação do modelo de programação CUDA. Os conceitos abordados neste capítulo são necessários para o embasamento da metodologia, desenvolvimento dos algoritmos e análises dos resultados dos testes, detalhados nos capítulos subsequentes.

# DESENVOLVIMENTO DOS algoritmos

Este capítulo detalha o processo de desenvolvimento dos algoritmos utilizados como foco dos testes no presente trabalho. Desta maneira, é possível a comparar o desempenho das execuções de cada um utilizando um modelo sequencial, um modelo baseado em OpenMP e outro baseado em CUDA.

Os testes são realizados utilizando diferentes parâmetros para a geração dos fractais: funções, dimensões da imagem, limite de iterações e número de *threads* (para os modelos paralelos). Ao final, compara-se o tempo necessário para a execução de cada teste, para cada modelo, avaliando-se o *speedup* obtido. Nas seções seguintes serão detalhados o ambiente de desenvolvimento e a metodologia utilizados para a realização das análises.

AMBIENTE DE DESENVOLVIMENTO E TESTES

Para a implementação dos algoritmos foi utilizada a linguagem de programação C/C++, com extensões para CUDA. O ambiente de desenvolvimento utilizado para a construção e testes dos algoritmos foi a IDE Microsoft Visual Studio 2010, em conjunto com o CUDA SDK versão 3.2 da NVIDIA. À época do desenvolvimento deste trabalho já havia sido disponibilizada a versão 4.0, porém, devido a problemas de compatibilidade com o *driver* de vídeo, optou-se por manter a versão 3.2. O software NVIDIA Parallel Nsight foi também utilizado para realizar a integração do CUDA SDK com o Visual Studio, de modo a simplificar e automatizar diversas etapas de configuração que teriam de ser executadas manualmente. Juntamente com o SDK é disponibilizado também o compilador NVCC, o qual é responsável pela geração do código executável no *hardware* da GPU.

O *hardware* utilizado durante o desenvolvimento dos algoritmos foi um laptop Sony VAIO VPCCW25FB, com processador Intel Core i3-330M de 2,13 GHz com dois núcleos físicos mais dois núcleos virtuais através da tecnologia HyperThreading; 4 GB de memória RAM (não foi possível obter detalhes sobre fabricante e latência de memória); sistema operacional Windows 7 Home Premium 64 bits; e placa gráfica NVIDIA GeForce GT 330M, com 512 MB de memória. A versão do *driver* de vídeo utilizada foi 8.17.12.6904, de 29 de junho de 2011. Os detalhes do dispositivo gráfico, exibidos no Quadro 5, foram obtidos através do programa Device Query, disponibilizado juntamente com o CUDA SDK, e algumas características adicionais foram obtidas de NVIDIA (2010a).



Quadro – Especificações técnicas da placa gráfica GeForce GT 330M.

Fonte: Elaborado pelo autor (2012).

Para o desenvolvimento dos algoritmos com OpenMP foi necessária apenas a inclusão da biblioteca *omp.h* e a adição do parâmetro de compilação */openmp*, visto que o compilador VC++ disponibilizado com o Visual Studio fornece suporte nativo à especificação 2.0 da API. Com isto, os *pragmas* utilizados para a especificação do paralelismo são reconhecidos e interpretados pelo compilador, permitindo a execução paralela dos algoritmos.

Embora CUDA proveja suporte nativo a OpenGL, a geração e exibição das imagens foi feita utilizando a biblioteca CImg. A escolha desta biblioteca deu-se pela praticidade de manipulação dos pixels, visto que apenas um arquivo de cabeçalho é necessário para utilizar as funções disponibilizadas pela biblioteca, e que os pixels são armazenados em um único *array* em *buffer*. Isto significa que, uma vez que a imagem é instanciada, todos os pixels são armazenados na memória ao mesmo tempo, podendo facilmente ser acessados através de ponteiros. Além disso, a imagem é estruturada linearmente, de modo que os canais de cores não são intercalados. Deste modo, a estrutura dos pixels da imagem segue a ordem R1R2R3Rn-G1G2G3Gn-B1B2B3Bn, e não R1G1B1-R2G2B2-R3G3B3-RnGnBn (onde R, G e B representam os canais de cores vermelho, verde e azul, respectivamente). Este conceito é ilustrado na Figura 19 a seguir:

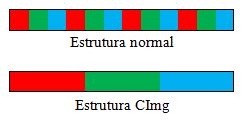


Figura – Organização de uma imagem normal e com a biblioteca CImg.

Fonte: Elaborado pelo autor (2012).

Maiores detalhes sobre a documentação da biblioteca podem ser encontrados em Tschumperlé (2004). É importante ressaltar que, embora a biblioteca CImg forneça diversas funções para tratamento de imagens, não fez-se uso destas neste trabalho. O foco manteve-se nos algoritmos responsáveis por calcular os fractais. Foi utilizada apenas uma função logarítmica, detalhada na seção seguinte, para colorir os objetos, de modo a prover uma melhor representação dos objetos e apresentar propriedades estéticas visualmente mais atrativas.

METODOLOGIA

Os fractais escolhidos para o desenvolvimento deste trabalho foram baseados no potencial de paralelização encontrado para cada método de geração e na disponibilidade e facilidade de acesso ao material na literatura. Optou-se por focar na geração de fractais mais simples, dado o tempo disponível para o desenvolvimento do trabalho e o fato de utilizar-se um modelo de programação ainda pouco familiar. Com isto, decidiu-se abordar os conjuntos de Julia e de Mandelbrot, descritos no capítulo 1. Para estes dois tipos de fractais foram utilizadas diferentes funções, escolhidas com base em dois critérios: as propriedades estéticas apresentadas nos objetos resultantes; e a possibilidade de avaliar o desempenho da GPU ao executar operações aritméticas não triviais. Assim, os testes para os conjuntos de Julia utilizarão as seguintes funções complexas:

()

()

()

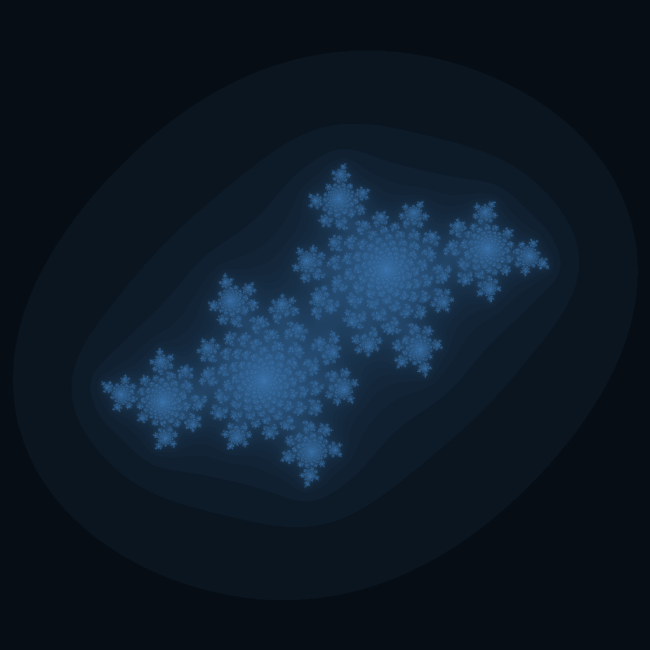
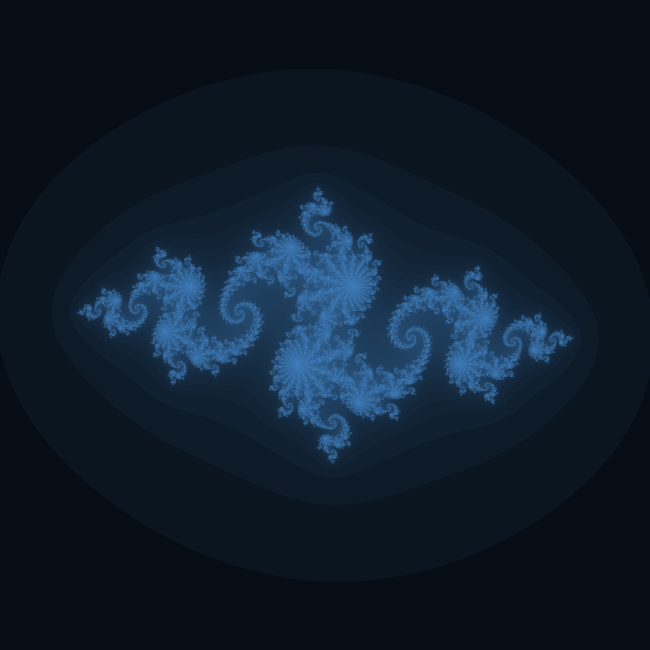
()

()

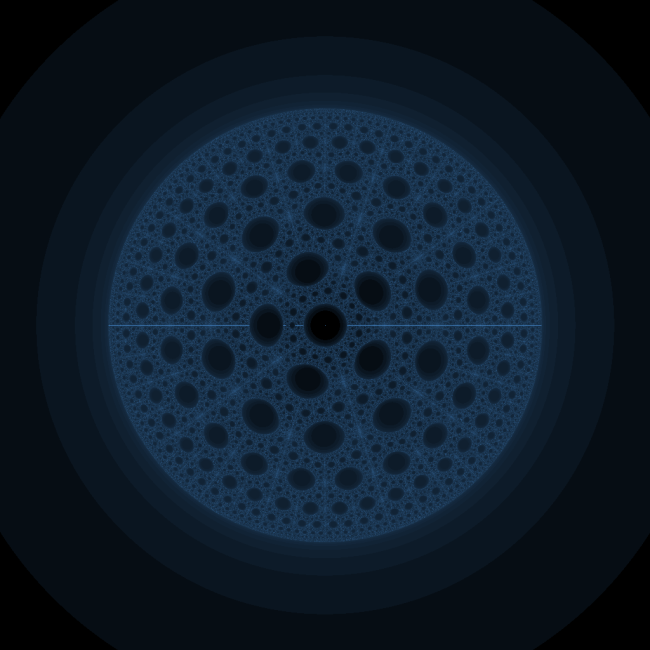
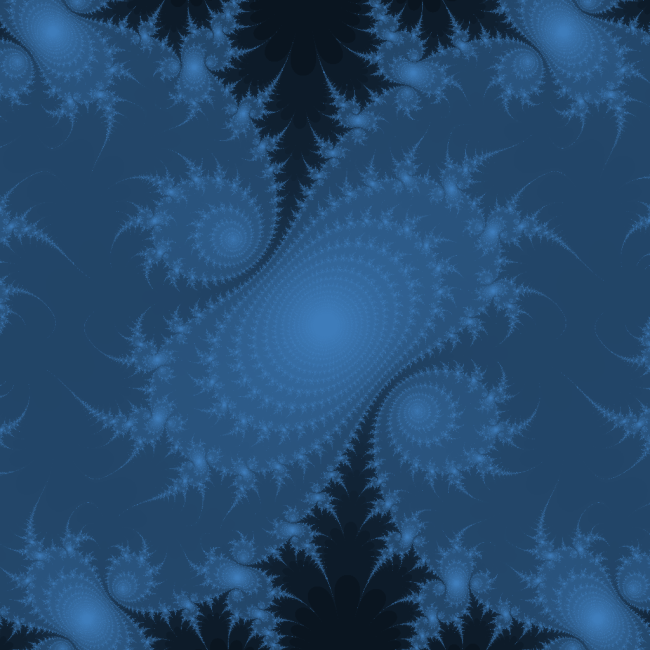
O conjunto de Mandelbrot, por sua vez, será analisado na sua forma mais comum, para a função quadrática descrita por

()

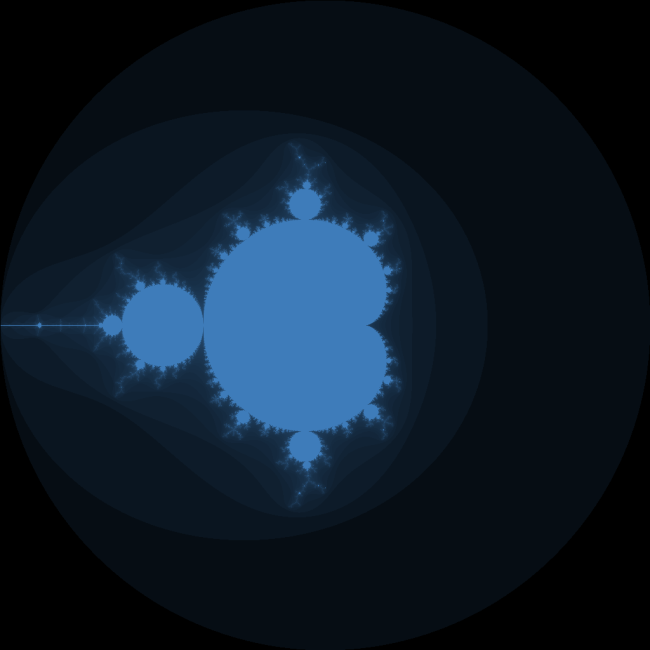
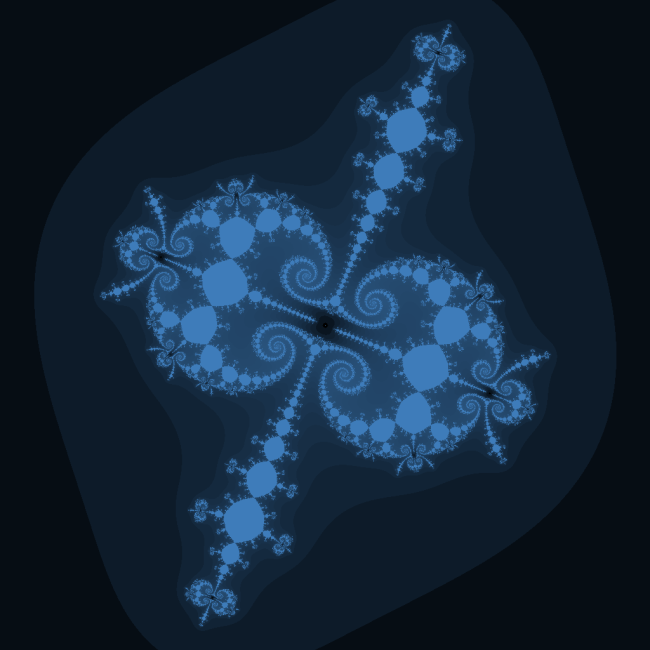
Estas funções dão origem aos objetos ilustrados a seguir na , onde as figuras (a) a (f) são resultantes das funções 9 a 14:



(a) (b)



(c) (d)



(e) (f)

Figura – Fractais resultantes das funções 9 a 14.

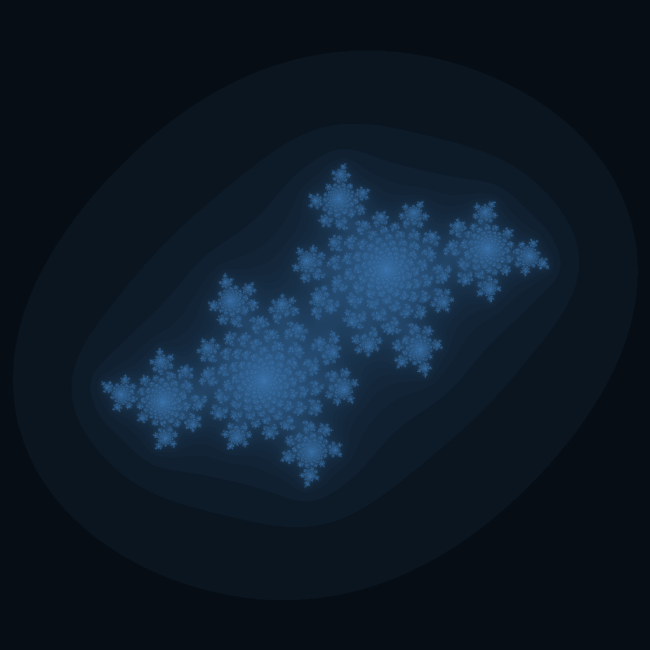
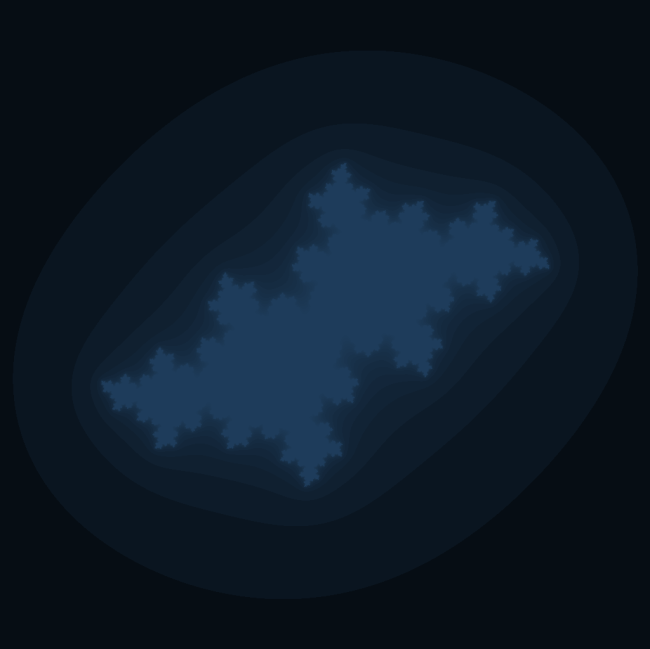
Fonte: Elaborado pelo autor (2012).

A representação dos números complexos utilizados para a geração dos fractais foi feita utilizando a biblioteca *complex*, para os testes baseados em CPU, e *cuComplex.h* para os testes com CUDA. A primeira disponibiliza sobrecarga de operadores e de funções definidas na biblioteca *cmath.h*, o que faz com que o código possa ser escrito da mesma maneira caso fossem utilizados números reais. A segunda biblioteca, entretanto, torna essa tarefa um pouco mais trabalhosa, visto que todas as operações aritméticas são realizadas na forma de funções, tornando a leitura do código menos intuitiva. Além disso, CUDA não disponibiliza uma função nativa para o cálculo de algumas operações com argumentos complexos como seno, o que é necessário para a execução dos testes referentes à função 11. Desta forma foi necessária uma implementação própria para realizar esta operação (ver Apêndice B).

Os algoritmos que realizam os cálculos dos fractais são executados utilizando-se diferentes valores para dois parâmetros: as dimensões da imagem e o número máximo de iterações para verificar se um pixel converge ou não ao atrator do fractal. O valor de escala é mantido em 1.

Para o primeiro, as imagens são exibidas utilizando dimensões 256 x 256, 512 x 512 e 1024 x 1024. Estas dimensões são adequadas especialmente para os testes com GPU, visto que estes valores, por serem múltiplos de 32 (tamanho de um *warp* de *threads*), permitem que se utilize a capacidade total de processamento do dispositivo. Conforme mencionado no capítulo 4, o *warp* é a unidade básica de escalonamento da GPU. Desta forma, a escolha destes valores foi feita de modo a possibilitar um alcance maior de desempenho nas execuções com CUDA.

O segundo parâmetro relaciona-se diretamente com o nível de detalhe exibido nas imagens resultantes, conforme conceituado no capítulo 1. Para tanto, 20, 100 e 500 iterações são utilizadas nos testes. A diferença no resultado final da imagem utilizando diferentes valores limites de iterações é ilustrada na Figura 21 abaixo:



1. (b)

Figura – Imagem gerada com limite de 20 (a) e 500 (b) iterações.

Fonte: Elaborado pelo autor (2012).

Os cálculos foram realizados com números de ponto-flutuante de precisão simples (32 bits), devido ao fato de a capacidade computacional do dispositivo utilizado é 1.2. Números de ponto-flutuante de precisão dupla são suportados apenas em dispositivos de capacidade computacional 1.3 e acima (NVIDIA, 2010a).

Com o intuito de prover propriedades estéticas mais interessantes, os fractais resultantes são coloridos extraindo-se o logaritmo natural do número de iterações necessário para verificar a divergência do pixel. Multiplica-se, então, o mesmo por um fator, diferente para cada um dos canais de cores da imagem, e extrai-se o resto da divisão por 255. Assim, evita-se a saturação da imagem com um número baixo de iterações e torna-se possível visualizar a dinâmica e o atrator do fractal a partir de trocas de cores sutis. A implementação deste algoritmo é disponibilizada em Apêndice.

Para efetuar a medição do tempo das execuções sequenciais e paralelas com OpenMP, foram utilizadas as funções *QueryPerformanceCounter()* e *QueryPerformanceFrequency()*, disponíveis com a biblioteca *Windows.h*, as quais utilizam contadores de alta precisão, com inteiros de 64 bits. A primeira função é usada para retornar o número de pulsos decorridos até o momento, enquanto a segunda retorna a quantidade de pulsos por segundo, número este que é dependente do *hardware* (MICROSOFT, 2012a). O tempo inicial é armazenado imediatamente antes das rotinas de geração dos fractais, e o tempo final imediatamente após as mesmas, de modo a garantir que apenas o tempo de processamento gasto nestas rotinas seja considerado.

Para as medições dos algoritmos com CUDA, foram utilizados os *timers* de eventos disponibilizados com a API, os quais possuem precisão de aproximadamente 500 nanossegundos (NVIDIA, 2010b). Dois cenários de teste são possíveis neste caso: um cenário que contemple o total de tempo gasto para alocar a memória na GPU e executar o *kernel*, e outro que considere apenas a segunda opção. Os testes contemplarão ambas as medições, de modo a identificar-se se o tempo necessário para a alocação de memória possui grande impacto nos testes realizados. Julgou-se que não é válido fazer comparações apenas usando o tempo de execução do *kernel* já que alocação e gerenciamento de memória são tarefas executadas implicitamente em testes executando apenas na CPU, e que afetam o tempo total de execução. Assim, para realizar comparativos mais realistas entre os modelos de execução baseados em CPU e GPU, as medições dos tempos de alocação e transferências de memória entre o dispositivo e o host também são contemplados, tarefas estas que devem ser realizadas explicitamente pelo desenvolvedor na arquitetura CUDA.

Os tempos apresentados para cada teste são o resultado da média de tempos de 20 execuções consecutivas, desconsiderando o tempo utilizado para a renderização da imagem, visto que esta tarefa não é o alvo de paralelismo deste trabalho, e apresentados em milissegundos. Estas médias, para os testes com OpenMP e CUDA, são utilizadas para calcular o *speedup* para cada configuração em relação à execução sequencial do modelo correspondente. Estes valores são calculados com base na fórmula abaixo, conforme Dongarra et al. (2003):

()

onde é o tempo da execução sequencial e o tempo de execução com *n* processadores. O *speedup* ideal é aquele que ocorre quando o tempo de execução da aplicação reduz de acordo com um fator inversamente proporcional ao número de processadores, de modo que, ao dobrar o número de processadores obtenha-se metade do tempo de execução. Na prática, contudo, fatores como coordenação de execução e distribuição de carga impedem que o *speedup* alcance o fator ótimo.

Para as execuções com CUDA, faz-se necessário liberar a memória utilizada para os cálculos após cada uma das 20 iterações. Isto evita a ocorrência de *overflows*, os quais fazem com que o dispositivo de vídeo deixe de responder, tendo de ser reiniciado.

Todos os testes e resultados apresentados neste trabalho foram realizados com a IDE Visual Studio configurada com a opção de compilação Release x64. Esta opção, ao contrário da opção Debug, faz com que o compilador realize diversas otimizações ao código (MICROSOFT, 2012b). Por consequência, tais otimizações permitem que os testes atinjam níveis de desempenho maiores, o que é apropriado para os propósitos deste estudo. Não serão apresentadas comparações de desempenho entre os testes compilados nas versões *Release* e *Debug*.

A análise dos testes baseados em OpenMP foi realizada modificando-se o parâmetro *schedule* nas diretivas para especificação do paralelismo, bem como os tamanhos dos *chunks* de *threads* para as iterações, conforme abordado no capítulo 2. Para os testes baseados em CUDA, as configurações de inicialização dos *kernels* foram feitas modificando-se o número de blocos e *threads*.

O capítulo seguinte descreve em maiores detalhes os testes executados para o desenvolvimento deste estudo, juntamente com a análise dos resultados dos mesmos.

# TESTES E análise dos resultados

Este capítulo é destinado a apresentar os testes executados para avaliar o desempenho dos algoritmos descritos no capítulo 5. Serão apresentados primeiramente os resultados das execuções sequenciais, seguidos pelos resultados obtidos através dos testes com OpenMP e, por fim, os resultados dos testes executados na GPU utilizando CUDA. Ao final, é feita uma comparação entre os resultados dos três modelos de teste.

Os quadros com os resultados dos tempos obtidos apresentados neste capítulo são estruturados de modo que as funções J(1) a J(5) representem os conjuntos de Julia gerados a partir das funções 9 a 13, respectivamente, apresentadas no capítulo 6. A função M(1), por sua vez, corresponde ao conjunto de Mandelbrot descrito pela função 14.

EXECUÇÕES EM MODO SEQUENCIAL

Os testes descritos nesta seção são referentes às execuções das funções descritas no capítulo anterior de modo sequencial. Isto implica que apenas um pixel da imagem é processado por vez. Dado o potencial de paralelização dos métodos de geração de fractais abordados, é natural imaginar que os algoritmos utilizando modelos OpenMP e CUDA apresentem um desempenho muito superior aos aqui analisados. Assim, os testes desta seção servem primariamente como base para análise do *speedup* obtido nos testes descritos nas seções subsequentes e para análise do comportamento das funções escolhidas em relação à dinâmica dos fractais.

O primeiro teste foi realizado através de dois laços aninhados percorrendo as duas dimensões da área mapeada à imagem. A cada pixel da imagem foram aplicadas repetidas iterações até que se verifique que o valor diverge ao infinito. Os resultados destas execuções são exibidos no Quadro 6:



Quadro – Médias de tempo obtidas para as execuções do primeiro teste no modo sequencial.

Fonte: Elaborado pelo autor (2012).

A partir destes dados é possível extrair algumas informações a respeito da dinâmica das funções em relação à variação dos parâmetros:

* Os tempos de execução aumentam quase linearmente à medida que as dimensões da imagem aumentam, comparando-se execuções com o mesmo número de iterações. Cada vez que as dimensões são dobradas, o número de pixels aumenta 4 vezes, e o tempo médio também é aumentado, na média, em um fator de 3,98. Este comportamento é ilustrado no Gráfico 1, utilizando-se a média aritmética dos tempos de execução para os 3 níveis de detalhe para cada função e dimensões de imagem. Com isto pode-se facilmente calcular o tempo aproximado de execução para outros tamanhos de imagem;
* Os fractais gerados a partir das funções J(1) e J(2) apresentam tempos de execução muito próximos para até 100 iterações. Ao se utilizar um limite de 500 iterações, entretanto, percebe-se que a função J(1) é mais custosa computacionalmente – em média 57,22% mais lenta que a função J(2). Ainda, a função J(1) apresenta um aumento no tempo de execução mais uniforme em relação a J(2), o que implica que o nível de detalhamento do fractal resultante possui relação com uma ampla faixa de valores de iterações. Os testes com 500 iterações são 61,01% mais demorados que as execuções com 100 iterações, enquanto estas são em média 88,37% mais longas que as com 20. A função J(2), entretanto, possui uma dinâmica mais relacionada a até 100 iterações, apresentando um acréscimo médio de 76% em comparação as execuções com limite 20 iterações, e de apenas 10% quando comparadas as execuções com 500 iterações em relação às de 100;
* A função senoidal J(3), apesar de composta por apenas duas operações (um seno e um multiplicação), é a mais onerosa para execuções com até 100 iterações, e a segunda mais onerosa com 500 iterações. Isto caracteriza um grande acréscimo no nível de detalhamento do fractal ao alterar-se o limite de iterações de 20 para 100, e um acréscimo moderado ao novamente aumentar o limite de 100 para 500. Esta conclusão é corroborada pelo expressivo percentual médio de aumento de 174,22% para a primeira situação, e de 34,27% para a segunda;
* O fractal gerado a partir da função J(4) apresenta variação de poucos milissegundos ao se aumentar o número de iterações. Com isto é possível deduzir que grande parte da dinâmica do sistema acontece logo nas primeiras iterações do algoritmo, de modo que o nível de detalhe das imagens modifica-se muito pouco, mesmo havendo um aumento de 25 vezes no limite de iterações. O percentual médio de aumento de tempo ao alterar o limite de 20 para 100 iterações foi de 9,55%, e variando o limite de 100 para 500 iterações, de 4,61%;
* A função J(5) exibe um grau de detalhamento muito grande quando se aplica a esta um limite de 500 iterações por pixel, o que pode ser observado nos 14,74 segundos necessários para a geração do fractal em 1024 x 1024. Além disso, execuções com este limite de iterações são em média 186,3% mais longas que as que utilizam limite de 100, enquanto estas, por sua vez, são em média 106,16% mais demoradas que os testes com apenas 20 iterações. Isto demonstra que o fractal possui um comportamento complexo e que sua dinâmica ocorre em uma ampla faixa de iterações;
* O conjunto de Mandelbrot M(1) é o fractal menos custoso quando gerado com limite de 20 iterações, porém é o terceiro mais custoso quando este limite é aumentado para 500. Há uma grande discrepância entre os tempos gerados a partir de cada valor de limite de iterações. Quanto este limite é alterado de 20 para 100, ocorre um aumento médio de 136,45% no tempo de execução. Ao novamente alterar-se este limite para 500, o aumento é de expressivos 276,2%, comprovando, assim, a complexa dinâmica deste fractal.



Gráfico – O aumento das dimensões da imagem causa um aumento quase linear no tempo de execução.

Fonte: Elaborado pelo autor (2012).

Decidiu-se então substituir o uso da função *std::abs()* para o cálculo da divergência dos valores por uma validação utilizando a função *pow()* aplicada às duas partes do número complexo z, dado que o cálculo do módulo de um determinado número complexo z na forma pode ser expresso por . Os resultados, apresentados no Quadro 7 a seguir, apresentam uma melhora significativa de desempenho em relação aos testes anteriores:



Quadro – Tempos obtidos para as execuções do segundo teste no modo sequencial.

Fonte: Elaborado pelo autor (2012).

Exceto pela função J(3), a qual apresentou uma redução média de apenas 1,39%, as outras funções tiveram reduções variando de 48,06%, caso da função J(5), a até 74,56% para a função J(1). Em média, obteve-se uma redução de 54,65%, considerando-se todos os resultados, o que demonstra a alta ineficiência da função *std::abs()* quando aplicada a argumentos complexos. Esta execução encerra a seção destinada à análise dos resultados obtidos com algoritmos sequenciais.

EXECUÇÕES COM OPENMP

Os testes detalhados nesta seção foram baseados nos mesmos algoritmos utilizados para os testes da seção anterior, apenas adicionando os *pragmas* para a especificação dos trechos de código a serem executados em paralelo. Dado que o processamento de cada pixel é feito de modo completamente independente dos outros, cada *thread* acessa diferentes endereços de memória, eliminando a necessidade de especificar regiões críticas dentro da seção paralela.

Considerar-se-á o segundo teste da seção anterior, o qual obteve maior desempenho, como base para os testes descritos nesta seção, bem como para o cálculo do *speedup*. De acordo com as especificações do processador utilizado, descritas no capítulo 5, há quatro núcleos de processamento dentro do chip. Assim, com base na fórmula 15, o *speedup* máximo teórico é um fator de quatro vezes. Ainda, o fato de haver quatro núcleos no chip faz com que o compilador OpenMP automaticamente defina quatro *threads* para a execução dos blocos de código paralelos. Este é o número de *threads* utilizado para os testes descritos a seguir.

O teste inicial foi feito adicionando o *pragma* *parallel for* antes do primeiro dos dois laços que varrem os pixels da imagem. A execução foi especificada com o parâmetro *schedule* configurado no modo *static*, sem especificar tamanho de *chunk*. Os resultados são apresentados abaixo no Quadro 8:



Quadro – Resultados das execuções com OpenMP com *schedule static*, sem definir tamanho de *chunk*.

Fonte: Elaborado pelo autor (2012).

Os tempos obtidos durante estas execuções representam um *speedup* variando de 1,49 a 2,75. Considerando as médias de todas as configurações para cada equação (médias extraídas para cada linha do quadro acima), as execuções para a função senoidal J(3) alcançaram o maior *speedup*, com um fator de 2,65, enquanto o conjunto de Mandelbrot M(1) obteve o menor, com um aumento médio de apenas 1,67 vezes. Ao considerar-se a média de todas as equações para cada diferente configuração (médias extraídas para cada coluna), o maior *speedup* é alcançado na geração dos fractais em tamanho 1024 x 1024 com limite de 20 iterações, consistindo num fator igual a 2,35. Considerando a média de todos os resultados, o *speedup* obtido foi um fator de 2,18 vezes.

Um segundo teste foi realizado alterando a configuração do parâmetro *schedule* para *dynamic*, também sem definir o tamanho do *chunk*. Conforme descrito no capítulo 2, quando o tamanho do *chunk* não é informado, o compilador define-o com o valor padrão igual a 1. Esta modificação proporcionou desempenho superior à configuração anterior, o que pode ser percebido ao se analisar os dados exibidos no Quadro 9:



Quadro – Resultados das execuções com OpenMP com *schedule dynamic*, sem definir tamanho de *chunk*.

Fonte: Elaborado pelo autor (2012).

Neste teste, os *speedups* em relação à execução sequencial variaram entre 2,08 e 3,27. Diferentemente da execução anterior, ao considerar-se as médias para cada função, o maior *speedup* foi alcançado para a função J(5), um fator de aumento de 3,11. A função M(1), entretanto, novamente obteve o menor ganho de desempenho, com um *speedup* de 2,60. Entretanto, este fator é muito próximo ao maior *speedup* obtido nas médias das execuções anteriores, o que demonstra que esta configuração, para a geração dos fractais escolhidos para este estudo, caracteriza uma abordagem mais eficiente do que o uso do parâmetro *schedule* com valor *static*. Ao extrair-se as médias de todas as equações para cada diferente configuração, a combinação de tamanho 1024 x 1024 e limite de 500 iterações apresentou o maior *speedup*, de 3,04.

Em relação à execução anterior, obteve-se *speedups* variando entre 1,05 e 2,17. Este último fator representa um aumento expressivo, alcançado para a execução de M(1) com uma imagem em 1024 x 1024 e 500 iterações. Em comparação com o teste anterior, onde esta equação obteve o menor *speedup*, aqui ela alcançou o maior, com um fator médio de 1,58.

Acredita-se que o aumento de desempenho causado ao se utilizar o valor *dynamic* pode ser justificado ao analisar-se a natureza do processo de geração dos fractais. Conforme detalhado anteriormente no capítulo 1, o processamento de cada pixel é feito até que uma determinada condição de divergência seja satisfeita, e naturalmente alguns pixels satisfarão esta condição mais rapidamente que outros. Assim, as *threads* que estiverem processando estes pixels terminarão suas tarefas mais cedo, e estarão disponíveis para executar mais um bloco de dados. Uma execução utilizando o parâmetro *schedule* com valor *static* fará com que as *threads* processem blocos contíguos de dados, o que implica que diversos pixels que poderiam estar sendo processados devem aguardar até que o bloco no qual estão contidos seja executado pela *thread* correspondente. O valor *dynamic* contorna este problema e delega novos blocos às *threads* tão logo estas finalizem suas tarefas atuais.

Tendo em vista o aumento de desempenho proporcionado pela configuração anterior, tentou-se verificar se fatores ainda maiores poderiam ser atingidos ao se definir a execução com um *chunk* para cada uma das quatro *threads*. Entretanto, os resultados não confirmaram as expectativas, com a maior parte deles sendo inferiores inclusive aos tempos do primeiro teste, com *schedule static*. Estes tempos são apresentados no Quadro 10:



Quadro – Resultados das execuções com OpenMP com *schedule dynamic* e um *chunk* para cada *thread*.

Fonte: Elaborado pelo autor (2012).

Embora as funções J(4) e M(1) tenham apresentado um desempenho ligeiramente superior, de 2,12 e 1,73, respectivamente, na média de todos os tempos obteve-se um *speedup* de 2,16, número inferior à média apresentada para os resultados no Quadro 8. Estes resultados mostram que a proposição feita na análise dos resultados ilustrados no Quadro 9 está imprecisa, contradizendo a afirmação de que execuções com o parâmetro schedule com valor *dynamic* alcançam níveis de desempenho superiores em relação às de valor *static*. Com base nesta configuração e nos tempos a partir dela obtidos, pode-se perceber que a simples definição do valor *dynamic* não oferece garantia de níveis maiores de desempenho para a geração dos fractais. O tamanho do *chunk* também deve ser definido adequadamente, conforme a natureza do problema de cada aplicação. Para os fins deste trabalho, *chunks* de tamanho muito grande aparentemente não são eficientes.

De posse desta informação, uma nova tentativa foi realizada mantendo-se o tipo de *schedule* como *dynamic*, porém definindo o tamanho do *chunk* igual a DIM/32, onde DIM refere-se à dimensão da imagem. Desta forma, o parâmetro *chunksize* possui valor 8 para imagens com tamanho igual a 256 x 256; 16 para imagens com tamanho de 512 x 512; e 32 para imagens com tamanho de 1024 x 1024. Com isto, esta configuração caracteriza um meio-termo entre as duas configurações anteriores. O ilustra os resultados obtidos para esta configuração:



Quadro – Resultados das execuções com OpenMP com *schedule dynamic* e *chunks* de tamanho DIM/32.

Fonte: Elaborado pelo autor (2012).

Os resultados apresentaram uma oscilação muito próxima àqueles apresentados no Quadro 9, de forma tal que alguns tempos foram ligeiramente menores e outros ligeiramente maiores. Os *speedups* em relação a esta execução variaram de 0,88 (o que na verdade não caracteriza *speedup*, mas sim uma perda de desempenho, dado que o fator é menor que 1) a 1,28. Na média por funções, a mais beneficiada foi a função M(1), com fator de 1,05. A função J(3) teve uma ligeira perda de desempenho, com fator de 0,99, enquanto J(5) obteve o mesmo fator. Contudo, no geral, considerando a média de todos os resultados, esta configuração proporcionou o maior *speedup*, alcançando um fator de 2,86 em relação à execução sequencial.

EXECUÇÕES COM CUDA

Assim como na seção anterior, os resultados aqui apresentados serão comparados com os apresentados no Quadro 7, no qual obteve-se o maior desempenho para execuções sequenciais. Para comparações com o modelo OpenMP, serão utilizados os resultados disponibilizados no Quadro 11, visto que estes apresentaram o maior *speedup* no geral.

Os testes focarão especialmente em diferentes configurações de *kernel*, com base nos conceitos já abordados no capítulo 4. Além disso, conforme justificado no capítulo 5, serão apresentados, para cada teste, o tempo de execução do *kernel* e o tempo total de processamento na GPU (considerando transferências de memória) discriminadamente. O *speedup* será calculado levando em consideração o tempo total de execução na GPU.

O primeiro teste executado foi invocando o *kernel* utilizando uma grade com DIM x DIM blocos, onde DIM equivale a uma das dimensões da imagem – 256, 512 ou 1024 pixels. Cada bloco é composto por apenas uma *thread*. Esta configuração sabidamente subutiliza a GPU, dado que cada bloco pode conter até 512 *threads* – conforme demonstrado no Quadro 5. Entretanto, deseja-se verificar se mesmo com tal configuração há algum ganho de desempenho nas execuções. Além disso, a imagem é alocada na memória global e é representada por um *array* unidimensional. Durante a execução do *kernel*, as *threads* escrevem os resultados de suas computações diretamente neste *array*. Os resultados são demonstrados no Quadro 12 abaixo:



Quadro – Resultados das execuções com CUDA utilizando DIM x DIM blocos com uma *thread*.

Fonte: Elaborado pelo autor (2012).

Um ponto requer uma explicação imediata ao iniciar a análise destes resultados: duas das execuções para a função J(3) não apresentam tempos de execução. Isto ocorre porque estas execuções são longas demais para serem concluídas, o que dispara o processo Watchdog[[2]](#footnote-2) do sistema operacional. De acordo com a NVIDIA (2012), o limite de tempo de execução de um *kernel* é de aproximadamente 5 segundos. Excedido este tempo, a aplicação para de responder e o *driver* de vídeo é reiniciado. Entretanto, os tempos destas duas execuções podem ser facilmente estimados com base nos resultados obtidos nos testes com fractais de tamanho 512 x 512, dado o crescimento quase linear já ilustrado no Gráfico 1. Assim, estima-se que, para 100 iterações, os tempos sejam de aproximadamente 4811,33 milissegundos, para o *kernel*, e um tempo total de 4824,70 milissegundos; e para 500 iterações, 8313,90 milissegundos para o *kernel* e 8353,83 milissegundos de tempo total. Nota-se que os tempos estimados para 100 iterações são próximos ao limite de 5 segundos de execução.

Em relação ao restante dos resultados obtidos, pode-se notar um *speedup* médio de apenas 2,08 em relação à execução sequencial utilizada como referência, com fatores variando entre 1,15 e 3,64. Na média por funções, o menor *speedup* ocorreu para a função J(3), com apenas 1,49, e o maior para J(5), com 3,35. Na comparação com os testes com OpenMP, houve *speedup* apenas para as execuções da função J(5), e de um fator ínfimo de 1,07. Para todas as outras funções houve redução de desempenho, sendo algumas das execuções até 151% mais lentas do que a execução correspondente com OpenMP.

Quanto ao tempo gasto com transferências de memória, percebe-se que este, na média, representou apenas 2,98% do tempo total de execução. Este tempo é mais representativo nas execuções com poucas iterações e tamanho de imagem reduzido, chegando a 5,82% na média para limite de 20 iterações e tamanho de 256 x 256. As execuções das funções J(3) e J(5), as quais notadamente demandam um tempo maior devido à natureza de seus cálculos, são as menos impactadas, de modo que 99,5% e 99,2% do tempo são gastos na execução do *kernel* para cada uma, respectivamente.

A comparação desta abordagem com os resultados obtidos a partir dos outros modelos permite perceber que um *kernel* com esta configuração é absolutamente ineficiente para estes cálculos. Possíveis justificativas para tal ineficiência incluem divergências no acesso à memória global; grande *overhead* gerado durante a execução do *kernel*; e o número de blocos sendo executados simultaneamente pelo *hardware* da GPU. Em relação à primeira, visto que cada bloco possui apenas uma *thread*, não é possível realizar operações de escrita coalescidas, já que cada *thread* necessita de um acesso individual à memória. Assim, o tráfego de operações na memória global é congestionado e o desempenho é comprometido. Para a segunda, dado que a execução das *threads* é organizada com base em *warps*, há um *overhead* muito grande no mecanismo de escalonamento da GPU, pois ao invés de 32 *threads* serem executadas, apenas uma é processada por bloco. Por fim, conforme descrito no capítulo 5, o dispositivo gráfico em uso permite a execução de até 8 blocos de *threads* por multiprocessador. Entretanto, para a configuração em uso, isto caracteriza apenas 8 pixels sendo processados por vez em cada núcleo.

Apesar destes motivos acredita-se que, a exemplo das execuções sequenciais já descritas, um aumento de desempenho pode ser alcançado simplesmente ao substituir o uso da função *cuCabsf()* – responsável por retornar o módulo de um argumento complexo para a validação de divergência dos pixels – por operações de multiplicação, antes mesmo de alterar os parâmetros de chamada do *kernel*. Esta modificação surtiu efeito, o que pode ser verificado no Quadro 13 a seguir:



Quadro – Resultados das execuções com CUDA utilizando DIM x DIM blocos com uma *thread* e cálculo de divergência utilizando a função *cuCabsf()*.

Fonte: Elaborado pelo autor (2012).

Novamente, duas das execuções para J(3) ultrapassaram o limite de tempo do Watchdog e não puderam ser concluídas. À parte disso, com exceção da função M(1), para a qual houve uma redução média de 27,68% no desempenho, todas as outras execuções foram mais rápidas que a execução anterior, com variações entre 1,01 e 2,03, e destaque para a função J(1), a qual obteve *speedup* médio de 1,68. Com isso, manter-se-á o uso da função *cuCabsf()* para M(1), enquanto para o restante das funções serão utilizadas operações de multiplicação normais.

De posse do aumento de desempenho demonstrado por estes resultados, os parâmetros do *kernel* foram então modificados de modo a alterar as dimensões e quantidades de blocos de *threads* e fazer melhor uso do poder de processamento da GPU. Conforme as especificações do dispositivo gráfico, descritas no Quadro 5, é possível que cada bloco possua um total de 512 *threads*. Assim, os blocos foram estruturados com dimensões de 32 x 16 x 1, de forma a totalizar o máximo de *threads* possível. A grade foi composta por DIM/32 x DIM/16 blocos, para que cada pixel seja processado por um *thread*. Como é possível que no máximo 1024 *threads* executem simultaneamente em cada multiprocessador, apenas 2 blocos de *threads* podem executar por vez, de um máximo de 8. O Quadro 14 exibe os tempos obtidos:



Quadro – Resultados das execuções com CUDA utilizando DIM/32 x DIM/16 blocos com 32 x 16 x 1 *threads*.

Fonte: Elaborado pelo autor (2012).

Os resultados obtidos demonstram uma melhora significativa em comparação aos testes anteriores. Em relação à configuração anterior, obteve-se *speedups* de até 32 vezes. Na média, esta configuração foi 15,15 vezes mais rápida que a anterior.

Aqui é possível perceber que o tempo necessário com transferências de memória constitui uma parcela muito maior do tempo total de execução, especialmente para as execuções com limite de 20 iterações. Na média destas execuções esta parcela equivale a 59,17% do tempo, chegando a 73,27% para a função J(2). As funções mais custosas computacionalmente – J(3) e J(5) – sofreram menor impacto com estas transferências, de modo que ao utilizar-se o maior valor de iterações e de dimensões da imagem a proporção foi de apenas 1,82% e 7,47% respectivamente. Isto demonstra o gargalo gerado pelo alto custo das transferências de memória à medida que o volume de tarefas executadas na GPU é reduzido. No caso de execuções com 256 x 256 pixels e limite de 20 iterações, o volume de tarefas a ser processado no dispositivo é tão baixo que o tempo de processamento do *kernel* é sobrepujado pelo tempo gasto com gerenciamento de memória. Naturalmente, situações como esta devem ser evitadas, pois subutilizam o poder de processamento da GPU. Idealmente, a GPU deve ter um volume de tarefas grande o suficiente para que transferências e latências de memória não representem parte significativa do tempo total de execução.

Em relação às execuções sequenciais, alcançou-se *speedups* de até 81 vezes, para a função J(5). Na média, esta função teve *speedup* de 57,23 vezes. Na média de todos os resultados obteve-se um *speedup* de 31,88 vezes. Comparando-se estes tempos com as execuções baseadas em OpenMP, alcançou-se um aumento de desempenho médio de 10,94 vezes, com fatores variando entre 3,9 e 25.

Utilizando este mesmo modelo, decidiu-se invocar o *kernel* utilizando o dobro de blocos, porém cada um com a metade do número máximo de *threads* permitido, de modo que 4 blocos possam executar em cada multiprocessador. Com isto, pretendeu-se verificar o comportamento do escalonador da GPU para controlar um maior número de blocos por vez, e analisar o possível impacto no desempenho. Assim, a grade foi montada com DIM/16 x DIM/16 blocos, e cada bloco contendo 16 x 16 x 1 *threads*, totalizando 256 *threads* por bloco. O Quadro 15 apresenta os resultados obtidos para esta configuração:



Quadro – Resultados das execuções com CUDA utilizando DIM/16 x DIM/16 blocos com 16 x 16 x 1 *threads*.

Fonte: Elaborado pelo autor (2012).

Esta configuração obteve uma ligeira melhora, na média, de 1,04 vezes em comparação com a configuração anterior, com fatores variando entre 0,82 e 1,17. Na média por funções, a função J(5) obteve um *speedup* de 1,08 vezes, enquanto a função J(2) não apresentou *speedup* em relação à configuração anterior, mantendo um fator de 1.

Em comparação com as execuções sequenciais, os *speedups* chegaram a fatores de até 85,73 vezes, com um fator mínimo igual a 10,15. Na média de todos os resultados, obteve-se um *speedup* de 33,43 vezes. A função J(5) apresentou um *speedup* médio de 62,13 vezes, enquanto M(1) apresentou um fator de 22,26. Na média por configurações, as execuções com 500 iterações apresentaram os maiores *speedups*, alcançando fatores de até 45,18 vezes. Já em relação às execuções com OpenMP, o *speedup* médio obtido foi de 11,46 vezes, em uma faixa de 4 a 26,83. O maior *speedup* foi novamente obtido para a função J(5), com média de 19,91, porém, diferentemente da comparação com as execuções sequenciais, aqui o menor *speedup* foi alcançado para a função J(2), sendo um fator de 7,62 vezes.

Por fim, uma última configuração foi utilizada de modo a aumentar a carga de trabalho de cada *thread*, fazendo com que estas processem 4 pixels, ao invés de apenas 1. Esta alteração foi feita com o intuito de se verificar a dependência do desempenho do dispositivo é mais fortemente associada ao número de *threads* no *kernel* ou à quantidade de operações executadas por cada uma delas. Para tanto, o *kernel* foi configurado de forma a totalizar DIM/4 *threads*, sendo a grade composta por DIM/32 x DIM/32 blocos e cada bloco por 16 x 16 x 1 *threads*. Os resultados são apresentados abaixo no Quadro 16:



Quadro – Resultados das execuções com CUDA utilizando DIM/32 x DIM/32 blocos com 16 x 16 x 1 *threads* e cada uma processando 4 pixels.

Fonte: Elaborado pelo autor (2012).

Esta configuração, ao analisar-se as médias de todos os tempos obtidos, não proporcionou *speedup* em relação à execução anterior, caracterizando um fator de 0,96 em relação a esta. Isoladamente, entretanto, algumas execuções apresentaram *speedups* de até 1,43 vezes. A redução mais significativa foi apresentada pela função J(5), a qual representou em média um fator de 0,88 vezes da configuração prévia. A função J(4) apresentou resultados em média iguais, enquanto M(1) alcançou um ligeiro aumento de 1,03 vezes.

Esta execução encerra a análise de desempenho dos algoritmos baseados em CUDA. Os resultados obtidos a partir dos testes realizados com os três modelos de programação e apresentados nas seções anteriores são discutidos na próxima seção. Desta maneira, pretende-se comparar os valores obtidos para justificar ou não a escolha por CUDA para a geração de conjuntos de Julia e de Mandelbrot.

DISCUSSÃO

Os primeiros resultados obtidos através das execuções sequenciais evidenciaram a complexidade do processamento dos fractais, com algumas execuções apresentando tempos de processamento acima de 10 segundos. Estes resultados, para cinco das seis funções analisadas, foram consideravelmente otimizados ao efetuar-se simples modificações nos cálculos de divergência dos pixels, apresentando reduções de mais de 74%.

De posse destes resultados, foram realizados os testes com o modelo OpenMP. Para as configurações de execução utilizando este modelo foram escolhidas apenas algumas poucas combinações, dentre muitas outras possíveis. Desta forma, naturalmente, é possível que haja combinações que produzam melhores resultados que os apresentados neste trabalho, vista a impossibilidade de testar cada uma delas dado o tempo disponível para o desenvolvimento do estudo.

Através dos resultados obtidos, entretanto, é possível perceber que as diferentes configurações de execução com OpenMP trazem variações de desempenho de acordo com a natureza do problema sendo tratado. No caso dos fractais este problema é diretamente perceptível nos objetos produzidos como resultado de cada função, visto que cada uma delas possui um comportamento completamente diferente das outras. Estas variações de *speedup* de acordo com a natureza dos problemas analisados são evidenciadas no Gráfico 2 a seguir. Idealmente, um ajuste fino deveria ser feito para cada uma das funções de modo a explorar suas peculiaridades e extrair o máximo de desempenho possível, utilizando diferentes tamanhos de *chunk* e diferentes tipos de *schedule* para cada uma delas.

Gráfico – *Speedups* obtidos a partir das execuções com OpenMP em relação às execuções sequenciais.

Fonte: Elaborado pelo autor (2012).

Este gráfico demonstra, por exemplo, que os fractais gerados a partir da função M(1) apresentam variações de desempenho muito mais fortemente relacionadas às dimensões da imagem e limite de iterações do que os fractais gerados a partir das funções J(3) e J(5). Além disso, o gráfico ilustra que os *speedup*s para as execuções com OpenMP são dependentes da combinação dos dois parâmetros. A função J(1), por exemplo, apresentou uma redução de *speedup* ao aumentar-se o número de iterações de 20 para 100 com a imagem em tamanho 256 x 256. Porém, com as dimensões da imagem em 512 x 512 e 1024 x 1024, os aumentos no limite de iterações proporcionaram aumentos também de *speedups*.

Através de uma configuração utilizando *schedule dynamic* com tamanho de *chunk* igual a DIM/32, obteve-se *speedups* de até 3,23 vezes, sendo o fator médio igual a 2,86. Julgou-se este valor satisfatório considerado que o processador utilizado consiste de dois núcleos físicos e mais dois núcleos virtuais habilitados através de *HyperThreading*. É possível que fatores de *speedup* mais próximos do máximo teórico de quatro vezes sejam alcançados ao fazer-se uso de processadores com quatro núcleos físicos, sem a utilização de núcleos virtuais.

Por fim, foram realizadas as execuções baseadas em GPU utilizando CUDA, as quais caracterizam o foco principal do estudo. Os resultados obtidos durante o desenvolvimento deste trabalho comprovaram as expectativas iniciais em relação ao aumento de desempenho proporcionado por esta arquitetura. Apesar disso, é imprescindível ressaltar que, em situações onde as configurações de execução dos *kernels* foram feitas de maneira inadequada, notou-se um aumento apenas irrisório no nível de desempenho dos algoritmos, que de modo algum justificaria a utilização de uma abordagem que faça uso de GPUs para o processamento de tarefas da natureza das apresentadas neste trabalho. Considera-se que fatores de *speedup* de apenas 1,15 vezes em relação à mesma execução feita de modo sequencial não justificam o esforço adicional necessário para a implementação de um algoritmo com CUDA, o qual inclui uma grande adaptação do código, gerenciamento explícito de memória, restrições de tempo de execução e disponibilidade de bibliotecas e funções reduzida.

Contudo, após alteradas as configurações dos *kernels*, de forma a utilizar quantidades de *threads* e blocos adequadas, notou-se uma melhora expressiva de desempenho. Os *speedups* alcançados chegaram a até 85,73 vezes, proporcionando uma redução no tempo de execução de aproximadamente 2,7 segundos para apenas 31,63 milissegundos. Considerando os resultados apresentados no Quadro 15, os quais representaram a execução mais rápida na média, obteve-se um *speedup* médio de 33,43 vezes. O Gráfico 3 a seguir ilustra os *speedups* obtidos através das execuções com CUDA:

Gráfico – *Speedups* obtidos a partir das execuções com CUDA em relação às execuções sequenciais.

Fonte: Elaborado pelo autor (2012).

As execuções com CUDA, em comparação com as execuções com OpenMP, apresentaram variações de *speedup* muito mais bruscas entre as diferentes combinações de parâmetros para a geração dos fractais. As funções que anteriormente apresentaram pouca variação de *speedup*, como J(3) e J(5), aqui apresentam variações de mais de 15 unidades em alguns casos. Além disso, é possível identificar padrões que se repetem durante toda a série apenas nas funções J(2), J(3) e M(1). A ausência de tais padrões no restante das funções torna difícil realizar projeções sobre o comportamento das mesmas utilizando maiores dimensões de imagem e limites de iterações mais altos que os utilizados durante este estudo. No entanto, esta ausência de padrões pode ser justificada devido ao fato de que, ao aumentar-se as dimensões da imagem, uma maior quantidade de pontos no plano complexo são mapeados e processados pelo algoritmo. Estes novos pontos, dependendo de suas localizações em relação ao atrator do fractal, podem divergir com números maiores ou menores de iterações. Especialmente se estiverem localizados próximos à borda dos fractais, pelos motivos descritos no capítulo 1, apresentarão um comportamento ainda mais imprevisível. Desta maneira é possível confirmar a proposição já feita previamente de que os fatores de *speedup* obtidos são dependentes da combinação dos dois parâmetros: dimensões da imagem e limite de iterações. O nível de escala, que durante este estudo não foi alterado, também é um parâmetro que apresentará influência nos fatores de *speedup* obtidos, pelo fato de que escalas implicam em uma maior precisão nos valores no plano complexo e, por consequência, possuem impacto no nível de detalhes do fractal resultante.

Um dos maiores limitadores do desempenho dos algoritmos com CUDA foi caracterizado pelas transferências de memória. Em algumas das execuções, o tempo gasto com estas transferências correspondeu a parcelas de até aproximadamente 75% do total. Os efeitos deste fato são perceptíveis ao se calcular o *speedup* obtido apenas com base no tempo de processamento do *kernel*. Nestes casos, os *speedups* alcançaram valores de até 118,35 vezes, e, em média, proporcionariam execuções aproximadamente 55 vezes mais rápidas em relação aos algoritmos sequenciais. Em comparação com os *speedups* obtidos considerando-se o tempo total de processamento, estes resultados são em média aproximadamente 39% mais rápidos. Estes valores, pelos motivos descritos no capítulo 5, são considerados apenas para efeito de comparação, visando ilustrar o gargalo representado pelas transferências de memória. Este gargalo é ilustrado também no Gráfico 4 a seguir:

Gráfico – Parcelas de tempo gastas com transferências de memória durante as execuções com CUDA.

Fonte: Elaborado pelo autor (2012).

Através deste gráfico é possível perceber que o uso da GPU para tarefas que envolvam um baixo volume de dados – neste caso, execuções com limite de 20 iterações – faz com que a parcela de tempo gasta efetivamente com o processamento dos *kernels* seja sobrepujada pelas transferências de memória. A utilização de uma GPU, portanto, é coerente para cenários onde haja um volume de dados suficientemente grande para que a parcela do tempo total gasta com alocação e cópia dos dados para a memória seja amortizada, caso de execuções com limites de iterações e dimensões da imagem mais altos.

Para *kernels* que necessitem de um tempo de processamento muito longo – caso das primeiras execuções com CUDA apresentadas neste capítulo, apesar de fazerem uso de configurações sabidamente impróprias – é possível dividi-los em múltiplas chamadas, mantendo os dados em memória. Desta maneira, o processo Watchdog não é disparado. Esta divisão não foi feita nos testes deste estudo com a intenção de demonstrar as implicações do processo Watchdog em aplicações que façam uso de GPUs como ferramentas computacionais de propósito geral. Naturalmente, o processo Watchdog não caracteriza um problema no caso de clusters de GPUs que não façam uso de vídeo.

Dada a natureza dos algoritmos utilizados nestes estudos, acredita-se que a parcela de tempo gasta com transferências de memória possa apenas ser suprimida com a utilização de um *hardware* com latências de memória menores, aumento da taxa de transferência do barramento PCI-e e *clocks* mais altos para acesso à memória. Visto que o processo de geração de fractais abordado neste trabalho é uma tarefa que não apresenta reuso de nenhum tipo de informação, a alocação e cópia dos dados para a memória do dispositivo caracterizam tarefas que devem ser realizadas uma vez para cada chamada de *kernel*. Além disso, visto que a alocação dos dados é feita na memória global e que a geração dos fractais é realizada através de funções matemáticas, o desempenho é fortemente relacionado à capacidade dos núcleos de processamento, de modo que a utilização das diferentes memórias apresentadas no capítulo 4 não deva apresentar ganhos de desempenho.

Não foi o foco deste estudo a análise de medidas como eficiência e largura de banda de memória obtidas durante os testes, especialmente por dois motivos: o fato de não haver na literatura pesquisada uma definição clara para a quantidade de FLOPS necessárias, com base no *hardware* utilizado, para determinadas operações utilizadas nos algoritmos, como seno, multiplicação, divisão e logaritmo natural (para colorir os fractais); e pelo fato de que as operações de escrita dos resultados dos cálculos realizados nos *kernels* sempre terem de ser feitas na memória global, sendo este um limitador inerente à natureza dos algoritmos utilizados.

CONCLUSÃO

A utilização de GPUs como dispositivos computacionais de propósito geral tem representado uma forte tendência nos últimos anos, especialmente no meio científico. O poder de processamento destes dispositivos tem viabilizado a resolução de problemas complexos e que façam uso de um grande volume de dados. A geração de fractais, objetos geométricos frequentemente definidos como sendo autossimilares, independentes de escala e com um infinito nível de detalhes, é uma das áreas que usufrui do advento das GPUs. A natureza dos algoritmos utilizados para este fim torna-os perfeitamente adequados para o processamento no dispositivo gráfico, visto que necessitam de um grande volume de operações aritméticas aplicadas repetidas vezes, de forma iterativa, sobre um conjunto de valores iniciais. Desta forma, com o objetivo de verificar o ganho de desempenho alcançado a partir das execuções de algoritmos para geração de fractais baseados em GPU, utilizando a arquitetura e modelo de programação CUDA, em relação a execuções sequenciais e paralelas dos mesmos com processadores tradicionais, foi realizada a análise apresentada neste estudo. Os fractais em foco foram os conjuntos de Mandelbrot e de Julia, para os quais foram utilizadas um total de seis diferentes funções. Para os testes paralelos baseados em CPU foi utilizado o modelo de programação OpenMP.

A fundamentação teórica sobre fractais, modelos de programação paralela e a arquitetura CUDA apresentada nos primeiros capítulos e aplicada à metodologia serviu de sustentação para o desenvolvimento dos testes descritos nos capítulos subsequentes. Os conceitos abordados foram necessários para ilustrar os objetivos do trabalho e justificar as implementações realizadas.

O primeiro capítulo discorreu sobre os fractais, de modo a prover o conhecimento necessário para a compreensão do restante do estudo e embasamento de seus objetivos. Foram apresentadas definições e aplicações dos fractais, incluindo outras áreas que não relacionadas à geração de imagens. Em relação aos tipos de fractais, um enfoque maior foi dado aos conjuntos de Mandelbrot e de Julia, escolhidos para abordagem durante os testes. Todavia, alguns outros tipos famosos de fractais foram apresentados, de modo a possibilitar propostas de futuros trabalhos.

No segundo capítulo foram apresentados conceitos básicos de programação paralela e dois dos principais modelos utilizados na atualidade, o MPI e OpenMP, para o qual foi dada uma atenção especial por ser o utilizado durante o desenvolvimento dos algoritmos.

O terceiro e quarto capítulos referiram-se à fundamentação teórica sobre a utilização de GPUs como dispositivos de processamento de propósito geral, apresentando os principais modelos de programação. O destaque e grande foco deste trabalho foi o modelo de programação e arquitetura CUDA, para a qual foram abordados conceitos como *kernels*, hierarquia de *threads* e os diferentes tipos de memória disponíveis, fornecendo o embasamento necessário para o desenvolvimento e execução dos algoritmos.

Os resultados obtidos a partir dos testes, utilizando a placa de vídeo GeForce GT 330M, comprovaram que a geração dos fractais escolhidos baseada em GPUs apresenta desempenho em média aproximadamente 33 vezes superior em relação a execuções sequenciais em CPU, com algumas execuções atingindo fatores de *speedup* de até quase 86 vezes. Os resultados das execuções com OpenMP, por sua vez, apresentaram *speedups* em média aproximadamente 2,86 vezes superiores às execuções sequenciais. Estes números satisfizeram o objetivo geral do trabalho.

Entretanto, diversos fatores possuem influência direta sobre os níveis de desempenho atingidos. O primeiro deles refere-se ao comportamento de cada fractal, definido por sua função geradora. Os resultados apresentados demonstram grandes variações de desempenho entre as funções, com algumas apresentando *speedups* médios iguais a pouco mais de 22 vezes, e outras chegando a fatores iguais a 62. Estas discrepâncias são justificadas principalmente pelo fato de que cada pixel apresenta condições de divergência diferentes, conforme a função geradora de cada fractal. Além disso, é possível notar que os *speedups* mais significativos foram obtidos para as funções compostas por operações aritméticas computacionalmente mais pesadas, como o seno e divisão.

Um segundo fator tem relação com a configuração dos *kernels*, para as execuções com CUDA. *Speedups* satisfatórios somente foram alcançados ao se utilizar parâmetros de configuração dos *kernels* que fizessem uso apropriado dos blocos e *threads*, através de uma correta distribuição dos dados de entrada. Pôde-se perceber que execuções com *kernels* configurados de modo impróprio proporcionaram níveis de desempenho baixíssimos, com fatores de *speedup* de apenas 1,18 vezes. Tais execuções, na média, apresentaram resultados inferiores aos obtidos com OpenMP.

Um terceiro fator com influência nos *speedups* alcançados é caracterizado pelas dimensões e nível de detalhes da imagem, definido pelo limite de iterações para cada fractal. A utilização de valores muito baixos para estes parâmetros fez com que o volume de dados das execuções fosse insuficiente para que os algoritmos desfrutassem do poder de processamento da GPU. Desta forma, os tempos de execução dos *kernels* foram sobrepujados pelas transferências de memória entre o host e o dispositivo.

Os principais obstáculos encontrados durante o desenvolvimento do trabalho foram caracterizados no âmbito da pesquisa sobre fractais. O alto nível de formalismos matemáticos presente na literatura referida evidencia a complexidade do tema e dificultou a redação do texto de forma clara e objetiva. Além disso, diversas implementações de outros tipos de fractais, especialmente das classes IFS e *fractal flames*, foram testadas, visando prover uma maior abrangência de testes. Entretanto, a inclusão destes fractais tornaria o escopo deste estudo grande demais, inviabilizando sua realização.

Quanto às limitações à realização do trabalho, diz-se que uma delas refere-se à medição de eficiência dos algoritmos apresentados. A ausência destas medições impossibilita saber se os testes executados fizeram uso máximo da capacidade dos processadores. Além disso, o *hardware* utilizado caracterizou uma limitação pelo fato de não ser parte das gerações mais recentes de processadores e GPUs.

Por fim, pode-se afirmar que os temas abordados como foco deste estudo possibilitam que diversos outros trabalhos possam ser propostos para o futuro, dando continuidade a este. Os resultados obtidos podem servir como base para análises utilizando diferentes modelos de programação, incluindo os modelos descritos neste estudo. O uso de OpenMP pode ser substituído por MPI, enquanto a arquitetura ATI Accelerated Parallel Processing pode ser utilizada ao invés de CUDA. Além disso, podem ser realizados trabalhos com maior enfoque em diferentes tipos de fractais, avaliando se níveis de desempenho similares são obtidos. Representações dos conjuntos de Mandelbrot e de Julia, foco deste estudo, podem ser realizadas utilizando números hipercomplexos, como os quatérnios. Acredita-se que tais representações demandem um poder de processamento muito maior do que o necessário para os testes apresentados neste trabalho, podendo fornecer medidas interessantes de desempenho. Os fractais dos tipos IFS e *fractal flames* podem também ser o foco de novos trabalhos por possuírem diferentes algoritmos para geração, além de apresentarem uma grande riqueza de propriedades estéticas.

O uso de OpenGL pode ser explorado para a exibição das imagens, visto que CUDA provê suporte nativo a este padrão. Assim, um estudo pode ser realizado para avaliar o desempenho de algoritmos utilizando OpenGL e comparar com os resultados aqui apresentados, para os quais foi utilizada a biblioteca CImg.

Referências Bibliográficas

ADDISON, Paul S. **Fractals and chaos: an illustrated course**. Bristol: Institute of Physics Publishing, 1997. 256 p.

ADVANCED MICRO DEVICES. **AMD CAL programming guide**. 2010. Versão 2.0, revisão 2.03. Disponível em: <http://developer.amd.com/sdks/amdappsdk/assets/AMD\_CAL\_Programming\_Guide\_v2.0.pdf>. Acesso em: 07 nov. 2011.

\_\_\_\_\_\_. **AMD Accelerated Parallel Processing OpenCL programming guide**. 2011. Versão 1.3f. Disponível em: <http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD\_Accelerated\_Parallel\_Processing\_OpenCL\_Programming\_Guide.pdf>. Acesso em: 06 nov. 2011.

AMORIM, Cláudio Luís; BARBOSA, Valmir Carneiro; FERNANDES, Edil Severiano Tavares. **Uma introdução à computação paralela e distribuída**. Campinas: UNICAMP, 1988. 256 p.

BARNEY, Blaise. **OpenMP**. 2012. Disponível em: <https://computing.llnl.gov/tutorials/openMP/>. Acesso em: 28 mar. 2012.

BARNSLEY, Michael F. **Fractals everywhere**. 2. ed. San Diego: Morgan Kaufmann, 1993. 560 p.

BARNSLEY, Michael; HUTCHINSON, John E.; STENFLO, Örjan. **V-variable fractals and superfractals**. [ca. 2004]. Disponível em: <http://wwwmaths.anu.edu.au/~barnsley/pdfs/V-var\_super\_fractals.pdf>. Acesso em: 24 fev. 2012.

BECKER, Karl-Heinz; DÖRFLER, Michael. **Dynamical systems and fractals: computer graphics experiments with Pascal**. Cambridge: Cambridge University Press, 1989. 412 p.

BOURKE, Paul. **Julia set fractal (2D)**. 2001. Disponível em: <http://paulbourke.net/fractals/juliaset>. Acesso em: 08 mar. 2012.

BRADLEY, Larry. **Chaos & fractals**. 2010. Disponível em: <http://www.stsci.edu/~lbradley/seminar/attractors.html>. Acesso em: 28 fev. 2012.

BROTHERS TECHNOLOGY. **Fractal music**. Disponível em: <http://www.brotherstechnology.com/math/fractal-music.html>. Acesso em: 11 out. 2011.

BURTON, David M. **The history of mathematics: an introduction**. 7. ed. New York: McGraw-Hill, 2011. 816 p.

CHANDRA, Rohit et al. **Parallel programming in OpenMP**. San Francisco: Morgan Kaufmann, 2001. 232 p.

CHAPMAN, Barbara; JOST, Gabriele; VAN DER PAS, Ruud. **Using OpenMP: portable shared memory parallel programming**. Cambridge: The MIT Press. 2008. 353 p.

CRANE, Keenan. **Ray tracing quaternion Julia sets on the GPU**. 2005. Disponível em: <http://users.cms.caltech.edu/~keenan/project\_qjulia.html>. Acesso em: 25 fev. 2012.

DEVANEY, Robert L. **Chaos, fractals, and dynamics: computer experiments in mathematics**. Boston: Addison-Wesley, 1990. 181 p.

\_\_\_\_\_\_. Unveiling the Mandelbrot set. **Plus Magazine**, Cambridge, n. 40. 2006. Disponível em: <http://plus.maths.org/content/os/issue40/features/devaney/index>. Acesso em: 13 mar. 2012.

DONGARRA, Jack et al. **Sourcebook of parallel computing**. San Francisco: Morgan Kaufmann, 2003. 842 p.

DRAVES, Scott; RECKASE, Erik. **The Fractal Flame algorithm**. 2003. Revisado em 2008. Disponível em: <http://flam3.com/flame\_draves.pdf>. Acesso em: 10 jan. 2012.

FALCONER, Kenneth. **Fractal geometry: mathematical foundations and applications**. Chichester: John Wiley & Sons, 1990. 288 p.

FARBER, Rob. **CUDA Application design and development**. Waltham: Morgan Kaufmann, 2011. 315 p.

FRAME, Michael; MANDELBROT, Benoit; NEGER, Nial. **Fractal geometry**. Disponível em: <http://classes.yale.edu/fractals/>. Acesso em: 25 out. 2011.

GLASKOWSKY, Peter. **Intel’s Larrabee – more and less than meets the eye**. 2008. Disponível em: <http://news.cnet.com/8301-13512\_3-10006184-23.html>. Acesso em: 04 nov. 2011.

GREEN, Melinda. **The Buddhabrot technique**. Disponível em: <http://www.superliminal.com/fractals/bbrot/bbrot.htm>. Acesso em: 16 abr. 2012.

HART, John C.; SANDIN, Daniel J.; KAUFFMAN, Louis H. **Ray tracing deterministic 3-D fractals**. Computer Graphics, volume 23, número 3. 1989.

HELMBERG, Gilbert. **Getting acquainted with fractals**. Berlin: de Gruyter, 2007. 177 p.

KHRONOS GROUP. **The OpenCL specification**. 2011. Versão 1.1, revisão 44. Disponível em: <http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf>. Acesso em: 05 nov. 2011.

KIRK, David B.; HWU, Wen-mei W. **Programming massively parallel processors: a hands-on approach**. Burlington: Morgan Kaufmann, 2010. 258 p.

MANDELBROT, Benoit B. **The fractal geometry of nature**. New York: W.H. Freeman and Company, 1982. 469 p.

\_\_\_\_\_\_. **Fractals and scaling in finance: discontinuity, concentration, risk**. New York: Springer-Verlag, 1997. 561 p.

MICROSOFT. **About timers**. 2012. Disponível em: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms644900(v=vs.85).aspx>. Acesso em: 24 mar. 2012.

\_\_\_\_\_\_. **How to: set debug and release configurations**. 2012. Disponível em: <http://msdn.microsoft.com/en-us/library/wx0123s5.aspx>. Acesso em: 04 abr. 2012.

\_\_\_\_\_\_. **Hardware watchdog timers design specification**. 2006. Disponível em: <http://msdn.microsoft.com/en-us/windows/hardware/gg463324>. Acesso em: 01 mai. 2012.

MITCHELL, Kerry. **Fractal art manifesto**. 1999. Disponível em: <https://www.fractalus.com/info/manifesto.htm>. Acesso em: 11 out. 2011.

NVIDIA. **CUDA programming guide**. 2010. Disponível em: <http://developer.download.nvidia.com/compute/cuda/3\_0/toolkit/docs/NVIDIA\_CUDA \_ProgrammingGuide.pdf>. Acesso em: 01 nov. 2011.

\_\_\_\_\_\_. **CUDA C best practices guide**. 2010. Versão 3.2. Disponível em: <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA\_C\_Best\_Practices\_Guide.pdf>. Acesso em: 27 jan. 2012.

\_\_\_\_\_\_. **CUDA FAQ**. 2012. Disponível em: <http://developer.nvidia.com/cuda-faq>. Acesso em: 01 mai. 2012.

OPENMP ARCHITECTURE REVIEW BOARD. **OpenMP application program interface**. 2011. Versão 3.1. Disponível em: <http://www.openmp.org/mp-documents/OpenMP3.1.pdf>. Acesso em: 17 fev. 2012.

PEITGEN, Heinz-Otto; RICHTER, Peter H. **The beauty of fractals: images of complex dynamical systems**. Berlin: Springer-Verlag, 1986. 199 p.

QUINN, Michael J. **Parallel programming in C with MPI and OpenMP**. New York, NY: McGraw-Hill, 2003. 544 p.

READE, John B. **Calculus with complex numbers**. London: Taylor & Francis, 2003. 112 p.

SANDERS, Jason; KANDROT, Edward. **CUDA by example: an introduction to general-purpose GPU programming**. Upper Saddle River: Addison-Wesley, 2011. 290 p.

SEILER, Larry et al. **Larrabee: a many-core x86 architecture for visual computing**. 2008. Disponível em: <http://softwarecommunity.intel.com/UserFiles/en-us/File/larrabee\_manycore.pdf>. Acesso em: 25 out. 2011.

SMITH, Ryan. **Intel kills Larrabee GPU, will not bring a discrete graphics product to market**. 2010. Disponível em: <http://www.anandtech.com/show/3738/intel-kills-larrabee-gpu-will-not-bring-a-discrete-graphics-product-to-market>. Acesso em: 06 nov. 2011.

SNIR, Marc et al. **MPI: the complete reference**. 1996. Disponível em: <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>. Acesso em: 20 abr. 2012.

TSCHUMPERLÉ, David. **The CImg library**. 2004. Disponível em: <http://cimg.sourceforge.net/index.shtml>. Acesso em: 19 mar. 2012.

SPROTT, Julien C. **Strange attractors: creating patterns in chaos**. 1993. Disponível em: <http://sprott.physics.wisc.edu/fractals/booktext/sabook.pdf>. Acesso em: 28 fev. 2012.

WHITE, Daniel. **The unraveling of the real 3D Mandelbulb**. 2009. Disponível em: <http://www.skytopia.com/project/fractal/mandelbulb.html>. Acesso em: 16 abr. 2012.

WILKINSON, Barry; ALLEN, Michael. **Parallel programming: techniques and applications using networked workstations and parallel computers**. 2. ed. Upper Saddle River: Pearson Prentice Hall, 2005. 467 p.

XIAO, Henry. **Fractal audio coding**. 2005. Tese (Mestrado em Ciência da Computação), Queen’s University, Kingston, Ontario, 2005.

YAN, S.; MAX, N.; MA, K.-L. 2010. **Polygonal surface advection applied to strange attractors.** Disponível em: <http://vis.cs.ucdavis.edu/Ultravis/papers/162\_Polygonal\_Surface\_Advection.pdf>. Acesso em: 22 fev. 2012.

# APÊNDICE

APÊNDICE A – *KERNELS* UTILIZADOS PARA OS TESTES COM CUDA

/\* Modelo de implementação para o cálculo do conjunto de Julia para a função

z = z² + c. Para os propósitos deste trabalho a função foi parametrizada

para os demais fractais \*/

\_\_device\_\_ int computeJuliaGPU(int x, int y) {

const int limit = 100;

int counter = 0;

/\* limites do plano complexo, podem variar conforme o fractal \*/

float xmin = -2;

float ymin = -2;

float xmax = 2;

float ymax = 2;

cuFloatComplex c = {-0.8f, 0.156f};

float jx = SCALE \* (xmin + x \* (xmax - xmin)/DIM);

float jy = SCALE \* (ymax - y \* (ymax - ymin)/DIM);

cuFloatComplex z = {jx, jy};

while ((counter <= MAX\_ITERATIONS) && ((z.x \* z.x) + (z.y \* z.y) < limit)) {

counter++;

z = cuCaddf(cuCmulf(z,z), c);

}

return counter;

}

/\* Cálculo do conjunto de Mandelbrot para a função z = z² + c.

Diferentemente dos conjuntos de Julia, aqui os pontos da tela

são mapeados para a variável complexa c \*/

\_\_device\_\_ int computeMandelbrotGPU(int x, int y){

int counter = 0;

float xmin = -2;

float ymin = -2;

float xmax = 2;

float ymax = 2;

float jx = SCALE \* (xmin + x \* (xmax - xmin)/DIM);

float jy = SCALE \* (ymax - y \* (ymax - ymin)/DIM);

cuFloatComplex c = {jx, jy};

cuFloatComplex z = {0, 0};

while ((counter <= MAX\_ITERATIONS) && (cuCabsf(cuCmulf(z,z)) < 4)) {

counter++;

z = cuCaddf(cuCmulf(z,z),c);

}

return counter;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Primeiro kernel: um grid com DIMxDIM blocos. Cada bloco contém apenas

uma thread. Este kernel é utilizado no primeiro e segundo testes do trabalho.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_global\_\_ void kernel\_1(float \*img, int type){

int x = blockIdx.x;

int y = blockIdx.y;

int offset = x + y \* DIM;

float color = 0;

/\* a variável color contém o número de iterações necessárias para que o ponto

escape ao infinito \*/

if (type == JULIA)

color = computeJuliaGPU(x,y);

else if (type == MANDELBROT)

color = computeMandelbrotGPU(x,y);

/\* as imagens utilizando a biblioteca CImg são armazenadas seguindo

o padrão {R1,R2,R3,Rn..., G1,G2,G3,Gn..., B1,B2,B3,Bn}, onde

R = vermelho, G = verde and B = azul. Desta forma, a variável offset

é utilizada para posicionar o ponteiro da imagem img de acordo com o

tamanho da imagem \*/

img[offset + 0] = (int)(log(color)\*10)%255;

img[offset + DIM\*DIM] = (int)(log(color)\*20)%255;

img[offset + 2\*(DIM\*DIM)] = (int)(log(color)\*30)%255;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Segundo kernel: grids com menor número de blocos são criados, de modo que cada

bloco contenha N threads, cada uma processando um pixel.

Este kernel é utilizado no terceiro e quarto testes

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_global\_\_ void kernel\_2(float \*img, int type){

int x = blockDim.x \* blockIdx.x + threadIdx.x;

int y = blockDim.y \* blockIdx.y + threadIdx.y;

int offset = x + y \* DIM;

float color;

if (type == JULIA)

color = computeJuliaGPU(x,y);

else if (type == MANDELBROT)

color = computeMandelbrotGPU(x,y);

img[offset + 0] = (int)(logf(color)\*10)%255;

img[offset + DIM\*DIM] = (int)(logf(color)\*20)%255;

img[offset + 2\*(DIM\*DIM)] = (int)(logf(color)\*30)%255;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Terceiro kernel: grids com menor número de blocos são criados, de modo que cada

bloco contenha N threads, cada uma processando múltiplos pixels.

Este kernel é utilizado no quinto testes

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_global\_\_ void kernel\_tile(float \*img, const int DIM\_TILE\_X, const int DIM\_TILE\_Y, int type){

int x = blockDim.x \* blockIdx.x + threadIdx.x;

int y = blockDim.y \* blockIdx.y + threadIdx.y;

int offset;

for(int i = 0; i < DIM\_TILE\_X; ++i){

for(int j = 0; j < DIM\_TILE\_Y; ++j){

int point\_x = (x \* DIM\_TILE\_X) + i;

int point\_y = (y \* DIM\_TILE\_Y) + j;

float color;

if (type == JULIA)

color = computeJuliaGPU(point\_x, point\_y);

else if (type == MANDELBROT)

color = computeMandelbrotGPU(point\_x, point\_y);

offset = point\_x + point\_y \* DIM;

img[offset + 0] = (int)(log(color)\*10)%255;

img[offset + DIM\*DIM] = (int)(log(color)\*20)%255;

img[offset + 2\*(DIM\*DIM)] = (int)(log(color)\*30)%255;

}

}

}

As condições de divergência utilizadas nos testes com CUDA tiveram de ser adaptadas de modo a usar a biblioteca *cuComplex.h*. Esta biblioteca não disponibiliza operadores sobrescritos, de modo que os cálculos aritméticos são todos realizados através de funções, o que dificulta a leitura e compreensão do código. A implementação inicial foi feita utilizando a função *cuCabsf()*:

while ((counter <= MAX\_ITERATIONS) && (cuCabsf(cuCmulf(z,z)) < 100)) {

counter++;

z = cuCaddf(cuCmulf(z,z), c);

}

Posteriormente, as implementações foram feitas conforme detalhado a seguir:

* Funções J(1) e J(2):

while ((counter <= MAX\_ITERATIONS) && ((z.x \* z.x) + (z.y \* z.y) < 100)) {

counter++;

z = cuCaddf(cuCmulf(z,z), c);

}

* Função J(3). A função senoidal é descrita no Apêndice B:

while ((counter <= MAX\_ITERATIONS) && (z.y < 50)){

counter++;

z = cuCmulf(c, cuCsinf(z));

}

* Função J(4):

while ((counter <= MAX\_ITERATIONS) && ((z.x \* z.x) + (z.y \* z.y) < limit)) {

counter++;

z = cuCdivf(cuCaddf(cuCmulf(z, cuCmulf(z, cuCmulf(z, cuCmulf(z,z)))), c), cuCmulf(z, cuCmulf(z, z)));

}

* Função J(5):

cuFloatComplex factor1 = {1.0f, 0.0f};

cuFloatComplex factor2 = {0.15f, 0.15f};

cuFloatComplex factor3 = {-3.0f, 3.0f};

while ((counter <= MAX\_ITERATIONS) && ((z.x \* z.x) + (z.y \* z.y) < limit)) {

counter++;

z = cuCdivf(factor1, cuCaddf(cuCmulf(factor2, cuCmulf(z,cuCmulf(z,cuCmulf(z,cuCmulf(z,z))))), cuCaddf(cuCmulf(z,cuCmulf(z,z)), cuCmulf(factor3,z))));

}

* Função M(1) (aqui manteve-se o uso da função *cuCabsf()*):

while ((counter <= MAX\_ITERATIONS) && (cuCabsf(cuCmulf(z,z)) < 4)){

counter++;

z = cuCaddf(cuCmulf(z,z),c);

}

APÊNDICE B – FUNÇÃO SENOIDAL COMPLEXA PARA CUDA

A função para o cálculo do seno complexo para CUDA pode ser implementada utilizando-se as funções trigonométricas do seno e cosseno normais e hiperbólicos para números reais:

\_\_device\_\_ cuFloatComplex cuCsinf(cuFloatComplex c) {

cuFloatComplex tmp;

tmp.x = sin(c.x)\*cosh(c.y);

tmp.y = cos(c.x)\*sinh(c.y);

return tmp;

}

APÊNDICE C – ALGORITMOS PARA AS EXECUÇÕES SEQUENCIAIS E COM OPENMP

A principal função utilizada para os testes é apresentada abaixo, para o cálculo dos conjuntos de Julia da forma . O código apresenta algumas leves mudanças visto que a versão original foi parametrizada para o cálculo dos demais fractais. A definição para que a execução seja feita com OpenMP é feita passando o valor *true* para o parâmetro *boolean* *openmp*. A diretiva *parallel* foi modificada variando os valores de *schedule* e *chunksize*.

CImg<float> julia\_1(int width, int height, bool openmp) {

CImg<float> img(width, height, 1, 3);

int number\_iterations = 0;

const int limit = 100;

float point\_x;

float point\_y;

float xmin = -2;

float ymin = -2;

float xmax = 2;

float ymax = 2;

int i = 0;

int j = 0;

std::complex<float> c(-0.8f, 0.156f);

#pragma omp parallel for if(openmp) shared(img) private(i, j, point\_x, point\_y, number\_iterations) schedule(dynamic, 32)

for (i = 0; i < width; i++){

for (j = 0; j < height; j++){

number\_iterations = 0;

point\_x = (xmin + i \* (xmax - xmin)/height);

point\_y = (ymax - j \* (ymax - ymin)/width);

std::complex<float> z(point\_x, point\_y);

while ((number\_iterations <= MAX\_ITERATIONS) && ((pow(z.real(),2)) + pow(z.imag(),2)) < limit) {

number\_iterations++;

z = pow(z,2) + c;

}

int colors[] = {(int)(logf(number\_iterations)\*10)%255,

(int)(logf(number\_iterations)\*20)%255,

(int)(logf(number\_iterations)\*30)%255};

img.draw\_point(i,j,colors,1);

}

}

return img;

}

A condição de divergência básica utilizada inicialmente durante os testes é descrita pelo laço abaixo. Esta condição foi utilizada pelas funções J(1) e J(2).

while ((number\_iterations <= MAX\_ITERATIONS) && (std::abs(pow(z,2)) < 100)) {

number\_iterations++;

z = pow(z,2) + c;

}

Removendo a função *abs()*, obteve-se o seguinte:

while ((number\_iterations <= MAX\_ITERATIONS) && ((pow(z.real(),2)) + pow(z.imag(),2)) < 100) {

number\_iterations++;

z = pow(z,2) + c;

}

As demais condições para o restante das funções são descritas a seguir:

* Função J(3):

while ((number\_iterations <= MAX\_ITERATIONS) && pow(z.imag(),2) < 50) {

number\_iterations++;

z = c\*sin(z);

}

* Função J(4):

while ((number\_iterations <= MAX\_ITERATIONS) && ((pow(z.real(),2)) + pow(z.imag(),2)) < 100) {

number\_iterations++;

z = (pow(z,5) + c)/pow(z,3);

}

* Função J(5):

std::complex<float> factor1(1.0f, 0.0f);

std::complex<float> factor2(0.15f, 0.15f);

std::complex<float> factor3(-3.0f, 3.0f);

while ((number\_iterations <= MAX\_ITERATIONS) && ((pow(z.real(),2)) + pow(z.imag(),2)) < 100) {

number\_iterations++;

z = factor1/(factor2\*pow(z,5) + pow(z,3)+(factor3\*z));

}

* Função M(1):

while ((number\_iterations <= MAX\_ITERATIONS) && ((pow(z.real(),2)) + pow(z.imag(),2)) < 4) {

number\_iterations++;

z = pow(z,2) + c;

}

1. Segundo Barnsley (1993), um ponto em um atrator é chamado de periódico quando, após aplicado a este uma sequência de iterações, é possível retornar ao valor inicial. O número de iterações que satisfaz esta condição é definido como o período do atrator. [↑](#footnote-ref-1)
2. Conforme a Microsoft (2006), um temporizador Watchdog é um componente de *hardware* que executa determinadas ações uma vez que um temporizador expira, como, por exemplo, reiniciar o sistema. Diz-se que o Watchdog é disparado caso este não tenha sido reinicializado durante um determinado período. [↑](#footnote-ref-2)