UNIVERSIDADE FEDERAL DE MATO GROSSO
INSTITUTO DE FÍSICA
PROGRAMA DE PÓS-GRADUÇÃO EM FÍSICA AMBIENTAL

Arquitetura de processamento paralelo com unidade de processamento gráfico de propósito geral para aplicação em séries temporais de dados ambientais

RAPHAEL DE SOUZA ROSA GOMES

JOSIEL MAIMONE FIGUEIREDO

CUIABÁ, MT, julho de 2012

UNIVERSIDADE FEDERAL DE MATO GROSSO
INSTITUTO DE FÍSICA
PROGRAMA DE PÓS-GRADUÇÃO EM FÍSICA AMBIENTAL

Arquitetura de processamento paralelo com unidade de processamento gráfico de propósito geral para aplicação em séries temporais de dados ambientais

RAPHAEL DE SOUZA ROSA GOMES

Dissertação apresentada ao Programa de Pós-graduação em Física Ambiental e da Universidade Federal de Mato Grosso, como parte dos requisitos para obtenção do título de Mestre em Física Ambiental.

Orientador:
JOSIEL MAIMONE FIGUEIREDO

CUIABÁ, MT, julho de 2012

Dados Internacionais de Catalogação na Fonte.

G633a Gomes, Raphael de Souza Rosa.
Arquitetura de processamento paralelo com unidade de processamento gráfico de propósito geral para aplicação em séries temporais de dados ambientais / Raphael de Souza Rosa Gomes. -- 2012
56 f. : il. color. ; 30 cm.

Orientador: Josiel Maimone Figueiredo.
Dissertação (mestrado) - Universidade Federal de Mato Grosso, Instituto de Física, Programa de Pós-Graduação em Física Ambiental, Cuiabá, 2012.
Inclui bibliografia.

1. GPGPU. 2. GPU. 3. CUDA. 4. OpenCL. 5. paralelismo. I. Título.

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

UNIVERSIDADE FEDERAL DE MATO GROSSO
INSTITUTO DE FÍSICA
Programa de Pós-Graduação em Física Ambiental

FOLHA DE APROVAÇÃO

TÍTULO: **ARQUITETURA DE PROCESSAMENTO PARALELO COM UNIDADE DE PROCESSAMENTO GRÁFICO DE PROPÓSITO GERAL PARA APLICAÇÃO EM SÉRIES TEMPORAIS DE DADOS AMBIENTAIS**

AUTOR: **RAPHAEL DE SOUZA ROSA GOMES**

Dissertação de Mestrado defendida e aprovada em 23 de novembro de 2012, pela comissão julgadora:

Prof. Dr. Josiel Maimone de Figueiredo – Orientador
Instituto de Computação – UFMT

Profa. Dra. Claudia Aparecida Martins – Examinadora Interna
Instituto de Computação – UFMT

Prof. Dr. Maurício Fernando Lima Pereira – Examinador Externo
Instituto de Computação – UFMT

Agradecimentos

Gostaria de agradecer as pessoas do PGFA que fizeram parte de todo o processo de montagem dessa dissertação e em especial gostaria de agradecer algumas pessoas desse programa.

Aos professores Iramaia e Serginho que me mostraram uma nova área da ciência que é a Teoria da Complexidade, pela qual quero estudar muito ainda.

Aos professores Denilton e Paraná por estarem sempre cobrando nossos esforços para fazer sempre o melhor trabalho.

Aos meus colegas de mestrado que me auxiliaram quando precisei para a compreensão das disciplinas.

Ao Geison por me ajudar a entender e validar as implementações dos cálculos, e também pelas conversas que tivemos sobre as teorias, como aplicá-las e nas construções dos artigos.

Ao meu orientador por compreender e me ajudar na parte mais difícil para mim, que é a escrita.

E quero agradecer principalmente a minha filha Clara e a minha mulher Daniela, pois sem elas jamais conseguiria fazer o trabalho e com certeza jamais estaria aqui onde estou. Feliz em aprender sempre e sabendo que jamais saberei tudo.

Sumário

Lista de Figuras

Lista de Abreviaturas e Siglas

GPGPU	:	General-purpose computing on graphics processing units;
CUDA	:	Compute Unified Device Architecture
OpenCL	:	Open Computing Language
$\mathbb{Z}$	:	Números Inteiros
WG_t	:	Quantidade Total de *work-group*
D_i	:	Quantidade de *work-group* da dimensão i
WG_t	:	Quantidade Total de *work-group* suportada pela GPU
WI_i	:	Quantidade Total de *work-itens* da dimensão i
G_i	:	Quantidade total de blocos por *grid* da dimensão i suportada pela GPU
g_i	:	Quantidade de blocos por *grid* da dimensão i definida pelo programa
T_T	:	Quantidade de total de *threads* por bloco suportada pela placa
T_i	:	Quantidade de total de *threads* por bloco da dimensão i
SM	:	*Streaming Multiprocessor*
SP	:	*Streaming Processor*
SMP	:	*Symmetric Multiprocessor*
UMA	:	*Uniform Memory Access*
NUMA	:	*Nonuniform Memory Access*
COMA	:	*Cache-Only Memory Architecture*

Resumo

GOMES, R. S. R. Arquitetura de processamento paralelo com unidade de processamento gráfico de propósito geral para aplicação em séries temporais de dados ambientais. 2012. 56f. Dissertação (Mestrado em Física Ambiental), Instituto de Física, Universidade Federal de Mato Grosso, Cuiabá, 2012.

As pesquisas e simulações científicas na área ambiental utilizam informações captadas por sensores para validar as modelagens e representações do comportamento do ambiente estudado. O custo de processamento desse contexto tende a ser extremamente alto, tanto pela quantidade de informações captadas pelos sensores, quanto pela complexidade dos cálculos envolvidos nas modelagens. Este trabalho apresenta uma biblioteca para processamento paralelo em ambientes com processadores gráficos (GPGPU) com aplicação em séries temporais de dados ambientais. A solução apresentada contrapõe com outras soluções de processamento paralelo, como *clusters* e *grid*, por ser de baixo custo e tornar transparente os aspectos de gerenciamento e programação do paralelismo das instruções. Para validação da solução apresentada foram implementadas bibliotecas nos padrões CUDA e OpenCL, que são considerados os principais padrões para GPGPU. Para medição do tempo de processamento, em ambas as bibliotecas foram aplicados testes utilizando o cálculo da dimensão fractal.

Palavras-chave: GPGPU, GPU, CUDA, OpenCL, paralelismo, dados ambientais.

Abstract

GOMES, R. S. R. Parallel processing architecture with general purpose graphics processing unit application for time series environmental data. 2012. 56f. Dissertação (Mestrado em Física Ambiental), Instituto de Física, Universidade Federal de Mato Grosso, Cuiabá, 2012.

The research and scientific simulations in the environmental area using information captured by sensors to validate the modeling and representation of the behavior of the environment studied. The processing cost of this context tends to be extremely high, so the amount of information captured by sensors, as the complexity of the calculations involved in modeling. This paper presents a library for parallel processing in environments with graphics processors (GPGPU) with application in time series of environmental data. The solution presented in contrast with other solutions for parallel processing, such as textit clusters and textit grid, because of its low cost and make transparent aspects of management and programming of parallel instructions. To validate the solution presented libraries were implemented in CUDA and OpenCL standards, which are considered the main standards for GPGPU. To measure the processing time in both libraries tests were applied using the method of fractal dimension.

Keywords: GPGPU, GPU, CUDA, OpenCL, parallelism, environmental data.

1 Introdução

A aquisição de dados tem se tornado uma atividade constante, para conhecer, entender, estudar e prever fenômenos de diversas áreas que nos cercam, como ocorre nas áreas de economia, meteorologia, computação, entre outras.

Além da necessidade de compreensão dos fenômenos, nos últimos anos, um fator que impulsionou o aumento considerável da quantidade de dados disponíveis, é a difusão de instrumentos e sensores cada vez mais precisos e de baixo custo, assim instrumentos que antes eram utilizados somente no meio científico, estão difundidos em vários contextos, como ocorre com os dispositivos GPS e medidores de temperatura e umidade, presentes até mesmo em dispositivos comuns.

Com essa facilidade na aquisição de informações e dados a tendência é que a compreensão dos fenômenos seja facilitada. Contudo analisar e processar esses dados em um espaço de tempo cada vez menor, é uma necessidade, pois a compreensão dos fenômenos é um fator decisivo na tomada de decisões de empresas, governos e instituições acadêmicas, auxiliando assim no desenvolvimento da qualidade de vida das pessoas.

Em relação aos dados ambientais a quantidade de dados é muito significativa, pois vários sensores podem medir várias variáveis micrometeorológicas em um tempo curto, por exemplo, medir a temperatura a cada 10 segundos, gerando uma quantidade de 3153600 de dados por ano dessa variável, logo quanto mais variáveis medirmos maior será o incremento de dados armazenados.

Após a obtenção dos dados é preciso realizar processamentos sobre os mesmos, com o intuito de entender os fenômenos nos quais os dados representam. No contexto do Programa de Pós-Graduação em Física Ambiental (PGPFA) da UFMT, as modelagens são realizadas para entendimento de fenômenos relacionados com o Pantanal e Floresta Amazônica. Para isso, existem vários cálculos que demandam alto custo de processamento, por exemplo, o cálculo para obter a dimensão fractal, o expoente de Lyapunov, a análise de recorrência, entre outros.

Na maioria das vezes o tipo de processamento nesse contexto envolve cálculos que podem

ser realizados em paralelo, o que indica a utilização de soluções, como *clusters* de computadores ou GPU's, contudo para utilizar esses recursos é necessário ter conhecimentos de programação específicos à plataforma adotada e ao tipo de paralelismo, bem como da organização do algoritmo para que ocorra o aproveitamento das funcionalidades do hardware paralelo.

Assim este trabalho apresenta uma forma de lidar com os recursos das GPU's para cálculos científicos através de uma biblioteca, com o intuito de tornar transparente ao programador os detalhes específicos da programação paralela e da plataforma utilizada, o que torna encurta o tempo de implementação e de manutenção do código fonte. As bibliotecas foram utilizadas em dois *softwares*, tanto em CUDA quanto em OpenCL, sendo implementado o cálculo da dimensão fractal. Para a validação dos algoritmos foi utilizado os dados das equações de Lorenz.

1.1 Objetivo Geral

Este trabalho tem como objetivo geral criar e validar uma biblioteca que encapsule as conexões e configurações na utilização de GPU em dados ambientais, mais especificamente com o uso das linguagens CUDA e OpenCL, visando otimizar a implementação, manutenção e a diminuição no tempo de resposta para os cálculos das modelagens de fenômenos micrometeorológicos.

1.2 Objetivos Específicos

- Estudar o comportamento dos dados ambientais.
- Estudar análises de séries não-lineares.
- Definir a estrutura da biblioteca para aplicação em dados ambientais.
- Construir uma biblioteca para a linguagem CUDA.
- Construir uma biblioteca para a linguagem OpenCL.
- Implementar o cálculo de dimensão fractal em CUDA, OpenCL e sequencial como estudo de caso.
- Testar a biblioteca do CUDA.
- Testar a biblioteca do OpenCL.

- Comparar os resultados do CUDA com os resultados da implementação sequencial para validação dos mesmos.
- Comparar os resultados do OpenCL com os resultados da implementação sequencial para validação dos mesmos.
- Avaliar o desempenho da biblioteca em CUDA
- Avaliar o desempenho da biblioteca em OpenCL

1.3 Organização do Trabalho

Este documento está organizado da seguinte forma:

- **Capítulo 2:** Neste capítulo os aspectos relacionados ao comportamento dos dados ambientais, as formas de estudá-los e modelá-los, bem como compreender e entender o escopo do trabalho para a utilização de processamento de dados paralelo.
- **Capítulo 3:** Neste capítulo é apresentada toda a metodologia utilizada para o uso do processamento paralelo desenvolvido neste trabalho.
- **Capítulo 4:** Neste capítulo é apresentado os resultados dos testes, da utilização dos recursos desenvolvidos, bem como a discussão dos mesmos através de um estudo de caso.
- **Capítulo 5:** Neste capítulo é apresentada a conclusão e o trabalhos futuros.

2 *Revisão Bibliográfica*

2.1 Dados Ambientais

A difusão e o barateamento dos dispositivos computacionais fizeram com que a quantidade de dados obtida por sensores, e manipulada através de simulações científicas, atingissem facilmente a ordem de *terabytes* (BALDWIN *et al.*, 2003). O barateamento dos sensores permitiu sua difusão em diversos contextos e portanto o aumento de dados captados; juntamente com o aumento do poder computacional (MOORE, 1998), as simulações e experimentos puderam ser executados mais rapidamente e com mais repetições, gerando mais dados.

No contexto deste trabalho, os dispositivos mais utilizados são aqueles voltados para captar dados micrometeorológicos, que são coletados através de vários sensores efetuando medições de múltiplas variáveis simultaneamente (NEVES, 2011). Essas medições são realizadas na ordem de segundos, portanto uma grande quantidade de dados é armazenada para posterior análise, gerando assim uma série temporal para cada variável medida.

Para analisar as séries temporais são utilizadas técnicas específicas para esse tipo de informação, e a complexidade aumenta pelo fato de geralmente não apresentarem comportamento linear, ou seja, um componente aleatório está presente nos dados, por isso são aplicadas técnicas de análises de séries temporais não-lineares, algumas explicadas a seguir.

2.2 Análises de Séries Temporais não-lineares

A Teoria da Complexidade representa hoje um ramo de estudo relativamente novo, e dentro dessa ciência existem inúmeros conceitos e fórmulas que são utilizados para representar fenômenos naturais observados pelo homem. Um conceito importante é o da reconstrução do espaço de fase para identificação do atrator (NUSSENZVEIG, 2000) utilizando defasagens temporais (TAKENS, 1981) em uma série de dados também temporal. Entende-se o espaço de fase como uma representação gráfica onde a variável tempo é oculta, e o atrator representa o conjunto

para onde tendem as trajetórias. A Figura 1 mostra o atrator construído a partir das equações propostas por Lorenz (LORENZ, 1963).

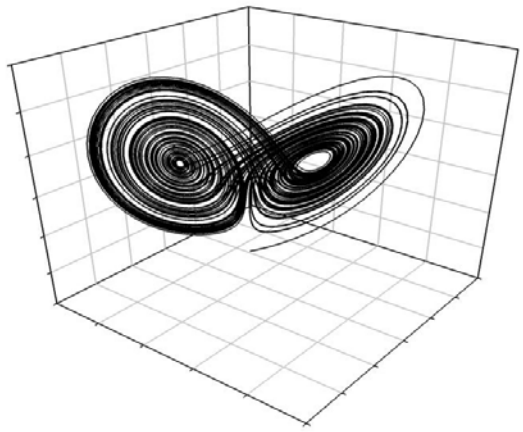


Figura 1: Atrator de Lorenz plotado a partir das variáveis x, y, z (MELLO, 2010).

A técnica de defasagem temporal utiliza os dados $X_0(t)$, e reconstrói as outras variáveis $X_k(t)$ (sendo k=1, ..., n-1). Depois aplica-se uma defasagem fixa τ (τ = mΔt, onde m é um número inteiro e Δt é um intervalo entre sucessivas amostras) para "n" pontos equidistantes do conjunto de dados. Isto é:

$X_0 : X_0(t_1), ..., X_0(t_n)$
$X_1 : X_0(t_1 + \tau), ..., X_0(t_n + \tau)$
:
:
:
$X_{n-1} : X_0[t_1 + (n-1)\tau], ..., X_0[t_n + (n-1)\tau]$

A escolha desse tempo de defasagem se deve a menor autocorrelação, na qual se obtem o maior ganho de informação da série de dados. Por esse fator é importante escolher o tao (τ) adequado como citado por (BAKER; GOLLUB, 1996).

2.2.1 Análise de Recorrência

Um comportamento fundamental de um sistema dinâmico determinístico é a recorrência de estados (THIEL, 2004). Um gráfico de recorrência possibilita a visualização dessa repetição, pois analisa o comportamento das trajetórias no espaço de fase (ECKMANN *et al.*, 1987).

O gráfico de recorrência ainda apresenta a vantagem de representar bidimensionalmente as repetições que ocorrem em espaços de fases maiores que três. Essa representação ocorre através de uma matriz definida como (ECKMANN *et al.*, 1987):

$$R_{i,j}(\varepsilon) = \theta(\varepsilon - ||X_i - X_j||), \quad i,j = 1,...,N \tag{1}$$

Onde N é a quantidade de dados, ε é a distância de sondagem e $\theta(.)$ é a função *Heaviside* (θ(x) = 0 se x $<$ 0, e θ(x) = 1 se x $>$ 0).

O gráfico é obtido plotando a matriz de recorrência resultante da equação 1, e utilizando diferentes cores para as entradas binárias, por exemplo, cor preta se $R_{i,j} \equiv 1$ e branco se $R_{i,j} \equiv 0$ (MARWAN *et al.*, 2007), pode-se então obter gráfico como a Figura 2 de (MARWAN, 2008).

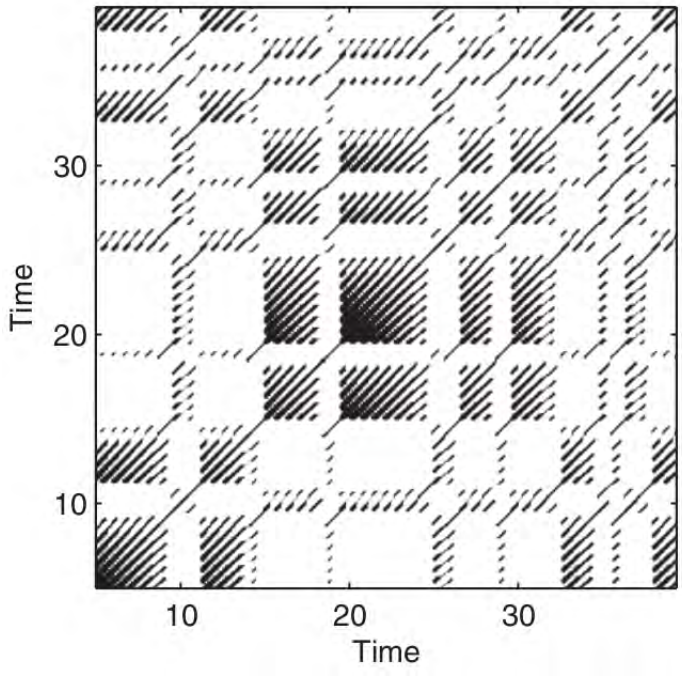


Figura 2: Gráfico de recorrência para o componente x da equação de Lorenz

A diagonal principal representa a série como um todo, as retas paralelas a essa diagonal representam a recorrência de uma trajetória em um tempo diferente.

2.2.2 Expoente de Lyapunov

O expoente de Lyapunov é muito utilizado para classificar o atrator, porque consegue mostrar a evolução da dinâmica do sistema, através da visualização de como as órbitas no atrator se movem, afastando-se ou aproximando-se (GOMES; VARRIALE, 2001), portanto o expoente quantifica a dependência desse movimento às condições iniciais do sistema (FIELDER-FERRARA; PRADO, 2011).

Considerando um sistema dinâmico n-dimensional em um espaço de fase, monitora-se a evolução de uma esfera infinitesimal de dimensão n. Essa esfera transforma-se em elipsoide, de dimensão também n, devido à deformação natural do fluxo da trajetória, portanto o expoente de Lyapunov da dimensão i (i= 1..n) é definida pelo tamanho dos eixos principais da elipse (WOLF *et al.*, 1985):

$$\lambda_i = \lim \frac{1}{t} \log \frac{p_i(t)}{p_i(0)} \tag{2}$$

Onde λ_i é o Lyapunov da dimensão i, $p_i(0)$ é o tamanho da esfera inicial e $p_i(t)$ é o tamanho do eixo principal da elipse. Pode-se classificar o atrator através do sinal dos expoentes de Lyapunov das dimensões (FIELDER-FERRARA; PRADO, 2011). Considerando um espaço tridimensional (n = 3), tem-se (Figura 3):

- Ponto Fixo: As trajetórias convergem para um determinado ponto (-,-,-).
- Ciclo limite: quando tem-se um expoente nulo (0,-,-) e o deslocamento do atrator acontece sobre uma trajetória.
- Toro T^2: o deslocamento acontece em duas direções (0,0,-).
- Atrator estranho: um expoente é positivo, outro nulo e um negativo (+,0,-).

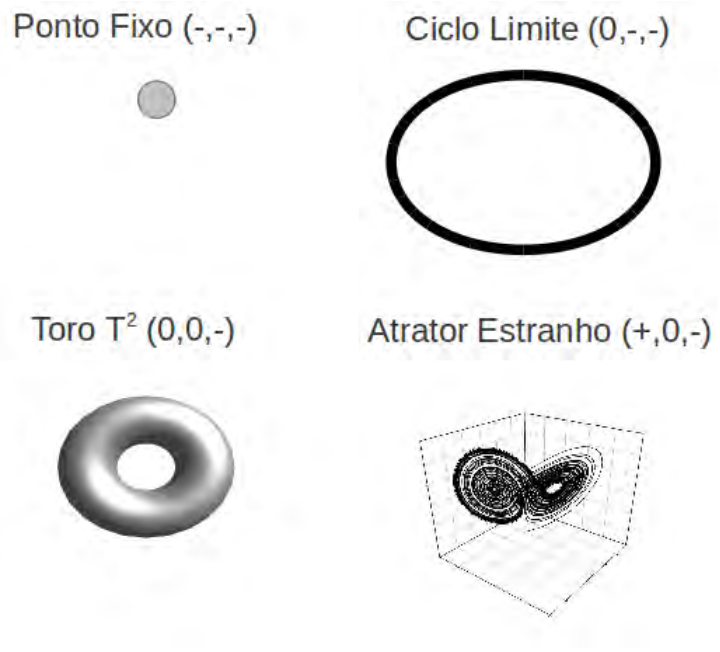


Figura 3: Representação das classificações dos atratores a partir do expoente de Lyapunov

2.2.3 Dimensão Fractal

A dimensão fractal é uma medida muito utilizada para medir a "estranheza"dos atratores, assim como o número de graus de liberdade e informações estatísticas sobre o sistema. Neste trabalho aplicou-se o algoritmo de (GRASSBERGER; PROCACCIA, 1983) que é um dos mais utilizados dentre os diferentes procedimentos desenvolvidos para computar a dimensão fractal. A dimensão de correlação (D_c) também provê o número de variáveis independentes necessárias para descrever a evolução temporal da dinâmica do sistema com a dimensão de imersão m (TAKENS, 1981) que tem como limite superior $2D_c$+1 para modelar um sistema.

Em um espaço de fase m-dimensional, a função correlação integrante $C(r)$ do atrator é dada por (GRASSBERGER; PROCACCIA, 1983):

$$C(r) = \frac{1}{n^2} \sum_{\substack{i,j=1 \\ i \neq j}}^{n} \theta(r - |X_i - X_j|) \tag{3}$$

Onde θ é a função *Heaviside*, θ(x) = 0 se x $<$ 0, e θ(x) = 1 se x $>$ 0. A partir de uma pequena correlação ε sonda-se a estrutura do atrator. Se esta estrutura é uma linha, o número de pontos dentro de uma sondagem a distância r de um ponto deve ser proporcional ao $\frac{r}{\varepsilon}$. Se for uma superfície, este número deve ser proporcional a $(\frac{r}{\varepsilon})^2$ e, de forma geral, se for uma dimensão d deve ser proporcional a $(\frac{r}{\varepsilon})^d$. Logo, para r relativamente pequeno, $C(r)$ deverá

variar conforme:

$$C(r) \approx r^d \tag{4}$$

Assim, como a dimensão de correlação (D_c) do atrator é aproximadamente igual à dimensão fractal d, ela é dada pela declinação $\ln(C(r))$ por $\ln(r)$ para um valor de r infinitesimal crescente até a integração total do atrator. Assim, a dimensão fractal é obtida do prolongamento linear da figura "joelho"(Figura 4A e 4B) do qual são extraídos os coeficientes lineares (Figura 4C).

$$\ln C(r) = d \ln r \tag{5}$$

Afere-se a dimensionalidade mínima, m, do espaço de fase dentro do qual o atrator mencionado está embutido. O parâmetro m define o número mínimo de variáveis que devem ser consideradas na descrição da dinâmica do sistema, ou seja, o número mínimo de equações diferenciais de primeira ordem que podem conter as características qualitativas do sistema dinâmico estudado (GRASSBERGER; PROCACCIA, 1983). Importante ressaltar que D_c é necessariamente menor que m.

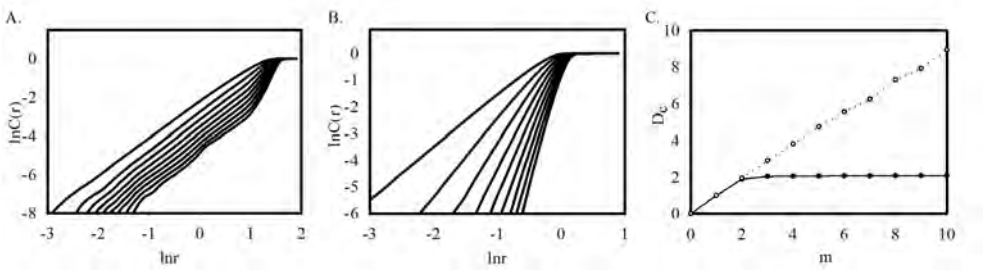
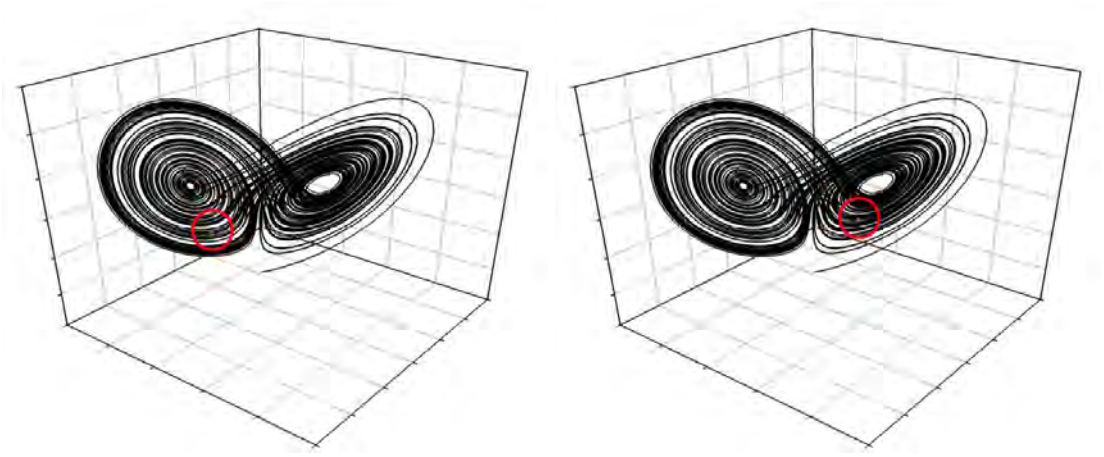


Figura 4: (a) $\ln C(r) vs \ln(r)$ para valores crescentes de m para o atrator reconstruído a partir da série x de Lorenz e (b) $\ln C(r) vs \ln(r)$ para uma série de dados aleatórios; e (c) saturação da dimensão de correlação versus dimensão de imersão m para o atrator reconstruído a partir da série x de Lorenz com $D_c \approx 2{,}05 \pm 0{,}01$ e $m = 3$ (•) e a instauração para a série aleatória com dimensão infinita (○).

As Figuras 5(a) e 5(b) mostram como acontece o cálculo dentro de um atrator. Para realizar o cálculo define-se um tamanho de raio r para uma esfera de dimensão d. Então para cada ponto do atrator verifica-se quantos pontos estão dentro da esfera. Esse cálculo é realizado para cada raio r e dimensão d definidos, portanto temos a soma das correlações para cada dimensão escolhida. Com a soma aplica-se a equação 5 resultando na Figura 4(a).



(a) (b)

Figura 5: Execução do cálculo da dimensão fractal dentro do atrator de Lorenz

Ao analisar a composição desse cálculo pode-se verificar que dependendo da quantidade de dados, do número de dimensões e de raios escolhidos, o tempo de resposta do algoritmo poderá ser muito grande. Uma possibilidade para otimizar esse tempo é utilizar a programação para GPU, denominado GPGPU (*General Purpose Computing on GPU*), sendo utilizado para este estudo as bibliotecas desenvolvidas para OpenCL e CUDA.

2.3 Processamento Paralelo

Processamento paralelo pode ser definido como sendo uma forma eficiente de processar informações que enfatiza a exploração de eventos no processo computacional (HWANG; BRIGGS, 1984). Esse processamento paralelo aparece de várias formas, *pipeling*, vetorização, simultaneidade, concorrência, paralelismo de dados, particionamento, multiplicidade, replicação, compartilhamento de tempo, compartilhamento de espaço, multitarefa, multiprogramação ou multicomputadores, entre outros (HWANG, 1993), sendo alguns desses explicados mais a frente.

Todas as formas de processamento paralelo demandam algum tipo de gerenciamento com o objetivo de sincronizar as entradas e saídas de cada unidade de processamento. Por exemplo, a Figura 6 mostra a arquitetura típica para servidores distribuídos, onde é necessário existir uma política de tolerância a falhas e portanto algum controle relacionado com replicação, falha de transmissão, entre outros. Na Figura 6 tem-se a tarefa a ser distribuída e o *dispatcher* que determinará para aonde a tarefa será enviada para o processamento. Após a finalização o resultado é encaminhado de volta para o *dispatcher* que por sua vez reenvia para a origem da tarefa.

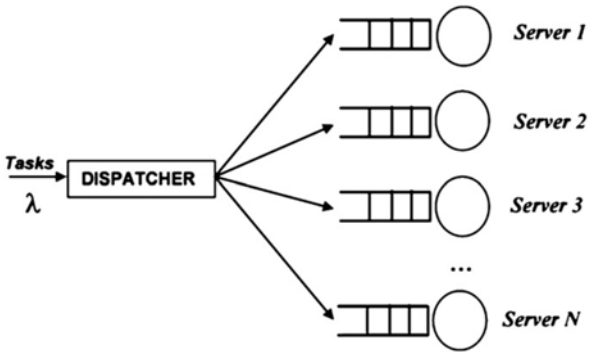


Figura 6: Modelo de servidor distribuído (SEMCHEDINE *et al.*, 2011)

2.3.1 Classificação da Arquitetura de Computadores

A arquitetura dos computadores pode ser classificada segundo a taxonomia de Flynn (FLYNN, 1972). Essa taxonomia propõem a arquitetura em duas dimensões: instruções e dados, sendo que cada dimensão possui dois valores: *single* ou *multiple*, com isso tem-se quatro classificações ilustradas nas Figuras 7, 8, 9, 10.

- *Single Instruction, Single Data* (SISD): Uma instrução é executada em um conjunto de dados. Exemplo: computadores mono *thread*.

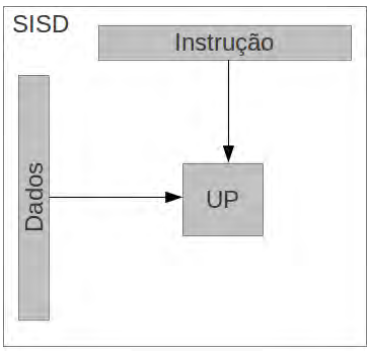


Figura 7: Representação de SISD

- *Multiple Instruction, Single Data* (MISD): Múltiplas instruções são executadas em um conjunto de dados. Exemplo: vários algoritmos de criptografia para decodificar uma mensagem.

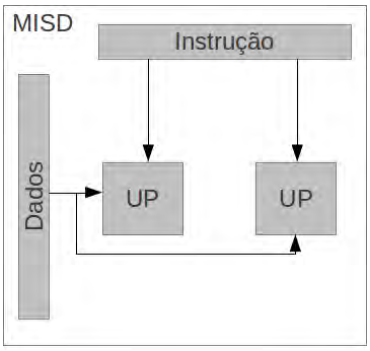


Figura 8: Representação de MISD

- *Single Instruction, Multiple Data* (SIMD): Uma instrução é executada em múltiplos conjunto de dados. Exemplo: processamento de imagens em GPU.

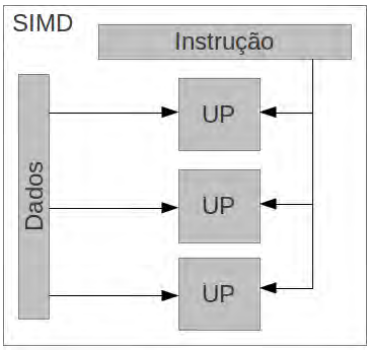


Figura 9: Representação de SIMD

- *Multiple Instruction, Multiple Data* (MIMD): Múltiplas instruções são executadas em múltiplos conjunto de dados. Exemplo: multiprocessadores e multicomputadores.

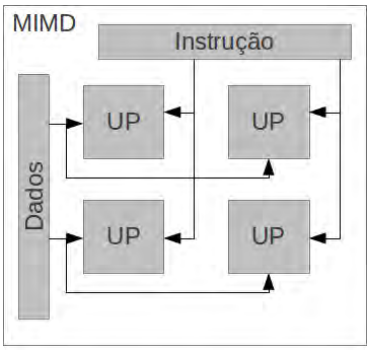


Figura 10: Representação de MIMD

Atualmente as GPU's são classificadas como SIMD, sendo que a programação para GPGPU é utilizada desde 1978, sendo amplamente utilizada no mundo científico atualmente, como pode-se ver em (WEIGEL, 2012; YAMANAKA *et al.*, 2011) entre outros.

2.3.2 CPU *Multinúcleos* e GPU

CPU *Multinúcleos* e GPU possuem em seu processador muitos núcleos, sendo o GPU voltado para paralelismo em dados (ELLIOTT; ANDERSON, 2012), enquanto CPU pode ter paralelismo de dados ou funcional. No tópico 2.4.5 os conceitos sobre GPU são detalhados. A figura 11 ilustra a divisão dos processadores e GPU's em núcleos (CPU). Atualmente os processadores podem atingir até 6 núcleos enquanto que as GPU's podem ter até 448 núcleos, com isso pode-se visualizar um grande potência para cálculos científicos devido a grande quantidade de processos ou *threads* que podem ser executadas simultaneamente.

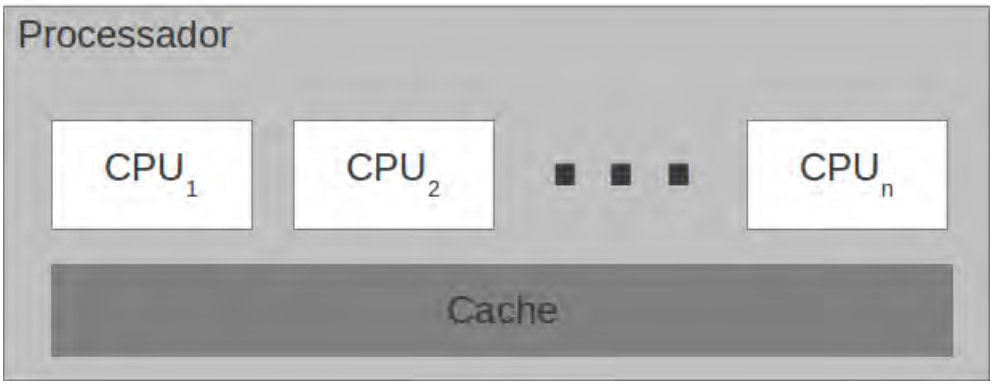


Figura 11: Divisão do processadores em núcleos

Outra característica importante dos multinúcleos ou *multi-cores* além de executaram de vários processos simultâneos é o baixo consumo de energia se comparado a um *single-core* (processador único). Isso acontece pelo fato dos *multi-cores* possuírem tempos de *clock* menores que os *single-cores*. Outro ponto é a menor produção de calor pelo fato de necessitarem de menos energia. Portanto, pode-se visualizar muito ganhos ao utilizar o processamento paralelo, contudo no presente estudo o fator analisado foi o ganho de desempenho sobre cálculos utilizando as GPU's em comparação com um *single-core*.

2.3.3 *Clusters*

Cluster pode ser descrito como a integração de mais de um computador e recursos incorporados através de hardware, redes e software para criar uma única imagem do sistema. Nas

abordagens tradicionais os termos de HPC (*High-Performance Computing*) e *cluster* de computadores referenciam o mesmo tipo de ambiente computacional (VALENTINI *et al.*, 2011).

Clusters são compostos por computadores sendo que cada um recebe o nome de nó. Cada nó pode ter diferentes características como a arquitetura de processador único ou múltiplo. Uma rede de computação em *cluster* é uma rede dedicada (VALENTINI *et al.*, 2011).

2.3.4 *Grids*

Grid é um conjunto de sistemas heterogêneos e autônomos de larga escala, de vários domínios administrativos, geograficamente distribuídos e interligados por uma ampla rede (TORKESTANI, 2011).

Os recursos da *grid* podem ser livremente adicionadas ou retiradas a qualquer momento a critério do proprietário. O desempenho dos nós da rede e sua carga frequentemente mudam com o tempo. *Grids* permitem a seleção, agregação e compartilhamento dos recursos de software e hardware de computadores diferentes em uma forma distribuída (TORKESTANI, 2011).

2.3.5 *Cloud Computing* (Computação em Nuvens)

Cloud Computing pode ser definido como um novo estilo de computação em que os recursos são dinamicamente escaláveis, muitas vezes virtualizados e são fornecidos como um serviço através da Internet (FURHT, 2010).

A principal diferença entre *cloud computing* e *clusters* é que o primeiro muitas vezes pode assumir a forma de ferramentas baseadas na web, e suas aplicações normalmente são acessadas através de um navegador de internet (VALENTINI *et al.*, 2011), tanto *grids* quanto *cloud computing* podem ser estruturados utilizando *clusters* (VALENTINI *et al.*, 2011).

2.4 Programação Paralela

A programação paralela consiste em desenvolver soluções utilizando o paralelismo suportado pelas máquinas atuais, assim pode-se obter um incremento na quantidade de processos executados simultaneamente. A programação paralela é desenvolvida de maneira diferente para cada tipo de arquitetura paralela: *cluster*, *grid*, *cloud*, GPU, etc. Pode-se obter o paralelismo dessas estruturas de várias formas:

- Aprender uma nova linguagem de programação voltada exclusivamente para essas estruturas como Occam, Ada ou HPF.
- Utilizar bibliotecas para programação paralela que utilize uma outra linguagem como hospedeira, como por exemplo, OpenMP, MPI, OpenCF, PVM, CUDA ou OpenCL.
- Utilizar compiladores que paralelizem o programa sequencial como por exemplo, Oxygen, OSCAR e PARADIGM para Fortran.

Para utilizar a programação paralela é necessário entender como modelar para paralelismo, pois a compreensão do problema é uma fator decisivo na determinação de quais partes podem ser paralelizadas. Além da modelagem entender como os processos são escalonados também é importante, pois o escalonamento pode influenciar diretamente na modelagem afim de se obter um maior ganho da estrutura que se está utilizando para o processamento paralelo. Outro fator é entender quais são os tipo de acesso à memória, pois pode-se deixar que determinados dados fiquem mais próximos daqueles que se relacionam frequentemente, assim diminui o tempo de latência entre um acesso e outro.

2.4.1 Metodologia de paralelização

O paralelismo tenta transformar um determinado problema, em problemas menores para que cada um possa ser processado de forma independente e paralela, otimizando o tempo de resposta. Para utilizar a programação paralela deve-se identificar e solucionar os problemas fundamentais do paralelismo (SARKAR, 1989):

- Identificar o paralelismo do problema;
- Particionar o problema em tarefas sequenciais e independentes;
- Dividir as tarefas em processos.

Ao resolver os problemas do paralelismo, teoricamente pode-se diminuir o tempo de resposta pela metade, se dividirmos o problema em dois. Contudo, é muito difícil atingir essa marca, pois a maioria dos problemas possui uma determinada parte que não pode ser subdividida, ou seja, permanece de forma sequencial. Com isso pode-se determinar o aumento de velocidade de um algoritmo pela lei de Amdahi (AMDAHL, 1967):

$$S = \frac{1}{r_s + \frac{r_p}{n}} \tag{6}$$

onde, S representa o ganho de velocidade, r_s é taxa sequencial do programa, r_p é a taxa paralelizável do programa e n é a quantidade de processos. Observe que $r_s + r_p = 1$. Ao analisar a lei de Amdahi, pode-se concluir que entender o problema é de fundamental importância na implementação de programas paralelos, pois se faz necessário conhecer qual parte é paralelizável e qual não é, para a partir dai verificar as possíveis tecnologias para paralelismo e escolher a que melhor lhe convém. Contudo, ainda há pouca utilização do paralelismo principalmente pela existência do problema da decomposição do algoritmo (HWANG; BRIGGS, 1984).

Para auxiliar o desenvolvimento de aplicações paralelas pode-se utilizar a metodologia de Foster (FOSTER, 1995), denominada PCAM, que é dividida em 4 partes:

- Decomposição/Partição (*Partition*): Esta etapa é destinada a realizar a divisão da tarefa em tarefas menores, de tal forma que tanto o cálculo quanto os dados associados a esse cálculo possam ser divididos.
- Comunicação (*Communicate*): A divisão das tarefas determinam o padrão de comunicação entre as tarefas, pois uma tarefa pode precisar acessar um dado que pertence a outra tarefa, assim é necessário determinar as estruturas e os algoritmos para realizar as comunicações entre as tarefas.
- Aglomeração (*Agglomerate*): As etapas anteriores eram etapas abstratas, portanto esta etapa tem por objetivo concretizar as abstrações anteriores, visando obter uma maior reutilização do código sequencial e diminuir o tempo de comunicação entre tarefas. Nesta etapa precisa-se também tomar o cuidado para não limitar a escalabilidade do algoritmo, por exemplo, se for utilizada uma matriz com as dimensões $512x256x2$, e a mesma for agrupada nas duas primeiras dimensões, ocorrerá uma limitação na escalabilidade em somente 2 processos ou processadores.
- Mapeamento (*Map*): Aqui é definida em qual processador as tarefas serão executadas para poder maximizar a ocupação dos processadores e minimizar sua comunicação.

A Figura 12 mostra todas as etapas da metodologia PCAM, onde o problema é analisado e particionado em tarefas menores, então é verificado quais tarefas se comunicam com quem, para então aglomerar aquelas que possuem muita comunicacão e finalizando distribuir as tarefas entre os processadores.

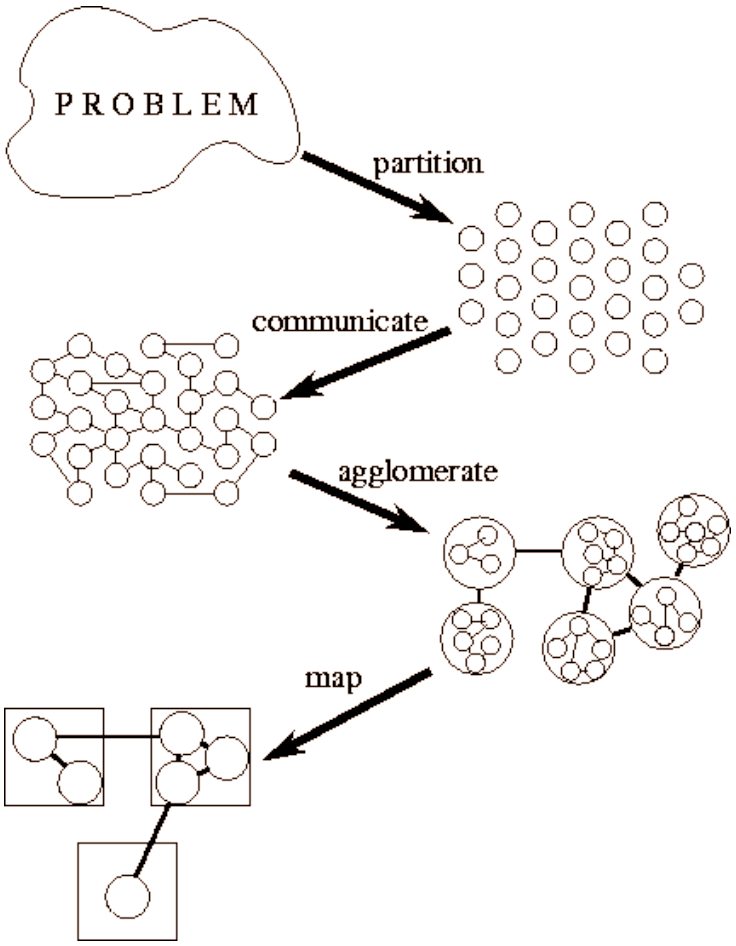


Figura 12: PCAM: uma metodologia de projeto para programas paralelos. Começando com uma especificação problema, desenvolvemos uma partição, determinar os requisitos de comunicação, aglomeração de tarefa e por último o mapeamento para os processadores (FOSTER, 1995).

2.4.2 Tipos de Paralelismo

Além da taxonomia de Flynn existem também alguns tipos de paralelismo.

- Paralelismo Funcional: Operações diferentes são executadas por tarefas diferentes em um conjunto de dados diferente.

Algoritmo 1 Algoritmo Funcional

1: $x = 1;$
2: $y = 1;$
3: $z = x + y;$
4: $w = x * y;$

No algoritmo 1 pode-se observar que as linhas 1 e 2 são independentes, e que as linhas 3 e 4 dependem das linhas anteriores, contudo são independentes entre si.

- Paralelismo de Dados: A mesma operação pode ser realizado sobre conjunto de dados diferentes.

Algoritmo 2 Algoritmo Dados

```
1: for i = 0 → tam_vetor do
2:     a[i] = b[i] + c[i];
3: end for
```

No algoritmo 2 observa-se que a soma dos dados pode ser realizada de forma independente.

- *Pipeline*: Divide o problema em etapas de forma que possam ser executados ao mesmo tempo, por exemplo uma linha de montagem. A figura 13 mostra um *pipeline* dividido em 4 fases: busca, decodificação, execução e escrita. Pode-se observar que depois do 4 ciclo de *clock*, a cada novo ciclo é obtido o resultado de uma execução, reduzindo assim o tempo de resposta.

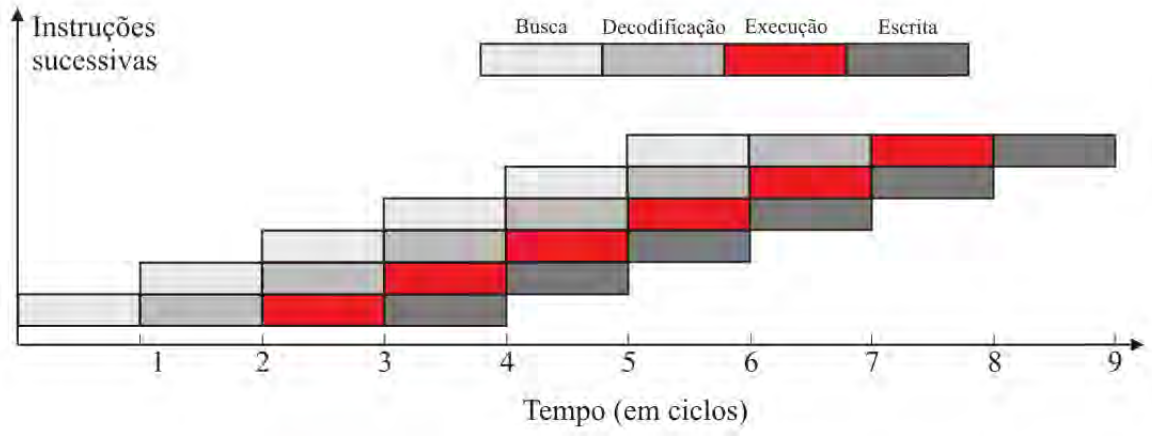


Figura 13: Execução em *pipeline* com uso de instruções sucessivas, onde o sistema passa a executar, após alguns ciclos, uma instrução por ciclo (PEREIRA, 2007).

2.4.3 Acesso à memória

Uma outra problemática da programação paralela é o acesso à memória, que pode apresentar diferentes formas, duas formas são:

- Memória Compartilhada: Os dados ficam armazenados em um local único, onde todos os processos podem ter acesso a esse determinado espaço, como mostra a Figura 14.

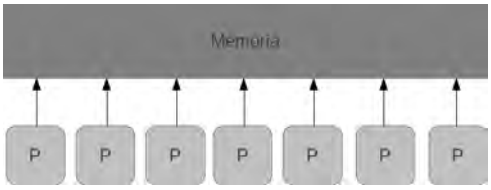


Figura 14: Memória Compartilhada

- Memória Distribuída: Nesse tipo de acesso cada processo possue um local de armazenamento de dados onde estes buscam as informações, como mostra a Figura 15.

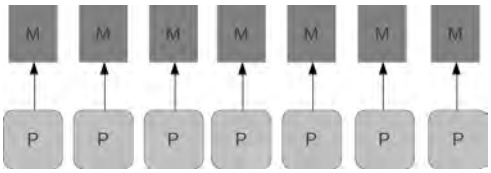


Figura 15: Memória Distribuída

Geralmente existem as duas abordagens na comunicação entre processos, devido ao fato de a memória local ter um acesso mais rápido e a memória compartilhada permitir que todos os processo utilizem os mesmos dados.

Classificação dos sistemas de memória compartilhada

A Figura 14 mostra que os processos P's podem acessar os dados da memória. Quando uma requisição de acesso a memória chega, esta é direcionada para o controlador da memória. Se aquele módulo que a requisição deseja acessar não estiver ocupado, o controlador responde a requisição permitindo o acesso e marca aquele módulo como ocupado. Se o módulo requerente estiver ocupado, o controlador emiti um sinal de ocupado para a requisição, fazendo com que o processo que enviou a requisição espere a liberação de acesso (HWANG, 1993). Segundo (HWANG, 1993), baseado neste processo de comunicação, pode-se classificar a memória compartilhada em:

1. *Uniform Memory Access* (UMA): A memória é acessível por todos os processadores através de uma rede de interconexão da mesma forma que um único processador acessa a memória. Todos os processadores têm tempo de acesso igual a qualquer local de memória. A interligação da rede usada em UMA pode ser um único barramento, barramentos múltiplos, ou um *switch*. Como o acesso à memória compartilhada é balanceado, esses sistemas são chamados de sistemas SMP (*symmetric multiprocessor*). Cada processador

tem igual oportunidade de leitura / gravação à memória, incluindo velocidades de acessos iguais.

2. *Nonuniform Memory Access* (NUMA): Cada processador liga-se a uma parte da memória compartilhada. A memória tem um único espaço de endereço. Portanto, qualquer processador pode acessar qualquer local de memória diretamente através do seu endereço real. No entanto, o tempo de acesso aos módulos depende da distância do processador.

3. *Cache-Only Memory Architecture* (COMA): Similar ao NUMA, cada processador liga-se a uma parte da memória compartilhada. No entanto, neste caso, a memória partilhada é a memória cache. O COMA requer a transferência dos dados para o processador requisitante. Não há hierarquia de memória e o espaço de endereço é feita de todos os caches.

2.4.4 Medidas de Desempenho

Existem basicamente dois tipos de medidas para performance, são o fator *speedup* e a eficiência (STAROBA, 2004), sendo que no presente trabalho foi utilizado o *speedup*. O *speedup* de um sistema paralelo pode ser definida como o razão entre o tempo gasto por um único processador para resolver um problema determinado, pelo tempo gasto por um sistema paralelo de n processadores para resolver o mesmo problema (HWANG, 1993).

$$S(n) = \frac{T_1}{T_n} \tag{7}$$

onde T_1 representa o tempo de execução com um processador e T_n representa o tempo de execução com n processadores.

Segundo (STAROBA, 2004) existem vários fatores que sobrecarregam programas paralelos limitando o aumento de velocidade e eficiência:

1. Períodos em que nem todos os processadores executam trabalho útil e são simplesmente ociosos.

2. Computação extra na versão paralela não aparece na versão sequencial.

3. Tempo de comunicação para envio de mensagens.

4. Agrupamento, criação e gestão de processos.

5. Sincronização dos processos.

2.4.5 *Graphical Processing Unit* (GPU)

As placas gráficas são componentes de hardware que determinam como as imagens serão apresentadas no monitor, sendo geralmente as informações armazenadas em uma memória interna da placa.

As placas gráficas são responsáveis por traduzir os binários enviados da CPU para uma imagem formada por *pixels* (o menor elemento de uma imagem que é possível atribuir uma determinada cor). Depois dessa tradução a placa envia os *pixels* para o monitor exibí-los, formando a imagem. Para realizar esse trabalho a placa gráfica é dividida em 4 partes:

- Conexão CPU: conecta a placa gráfica com a placa-mãe para realizar troca de dados com a CPU e obter energia;
- Processador: nesta parte define-se para onde os *pixels* serão enviados para o monitor;
- Memória: onde as informações sobre a imagem são armazenadas;
- Conexão Monitor: é por onde os *pixels* serão enviados para o monitor.

A placa gráfica ou placa de vídeo assemelha-se muito com uma placa-mãe. Ambas possuem memória e processador, contudo em uma placa de vídeo o processador recebe o nome de GPU (*Graphics Processing Unit*, ou Unidade de Processamento Gráfico), enquanto a placa-mãe utiliza o CPU (*Central Processing Unit*, ou Unidade de Processamento Central). A placa de vídeo possui um processador exclusivo pelo fato de precisar realizar cálculos com pontos flutuantes mais rapidamente. Para isso a placa já possui algumas funções específicas para estes cálculos. Outra diferença entre uma GPU e uma CPU é que a primeira pode possuir mais núcleos que uma CPU.

A arquitetura das GPU's circula em volta de um vetor escalável de *multi-thread*, *Streaming Multiprocessors* (SM) (NVIDIA, 2011), e cada *Streaming Processor* (SP) gerencia a alocação de memória, sincronização e a comunicação entre os SP's (PAPAKONSTANTINOU *et al.*, 2009), com isso o acesso ao paralelismo das GPU's fica facilitado. A Figura 16 mostra a divisão do SM's e SP's.

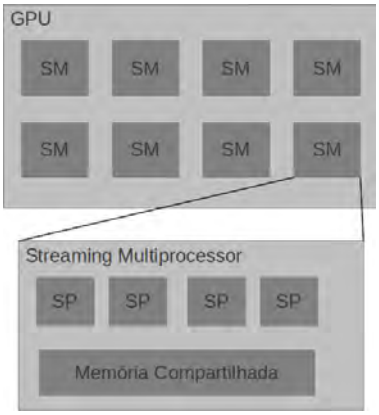


Figura 16: Modelo de programação GPU

Funcionamento da GPU

A realização de instruções na GPU acontece utilizando *pipeline* com muitos estágios, sendo que cada estágio é executado por um hardware específico paralelo. Contudo, existe a possibilidade dos estágios serem executados em um único hardware utilizando a unidade programável, essa arquitetura é denominada de Arquitetura *Shader* Unificada (OWENS *et al.*, 2008).

OpenCL

OpenCL é um padrão aberto para programação paralela de plataformas heterogêneas, que fornece um acesso de alto e baixo nível para processos em dispositivos paralelos. Funciona tanto em CPU's, quanto em GPU's de vários fabricantes (AUGUSTO; BARBOSA, 2012; NVIDIA, 2009a). OpenCL é também um framework que inclui uma linguagem, bibliotecas e uma API (GROUP, 2011). Cada dispositivo no OpenCL possui *p computing unit* (CU), que por sua vez é composta por *q processing elements* (PE), como mostra a Figura 17 (AUGUSTO; BARBOSA, 2012).

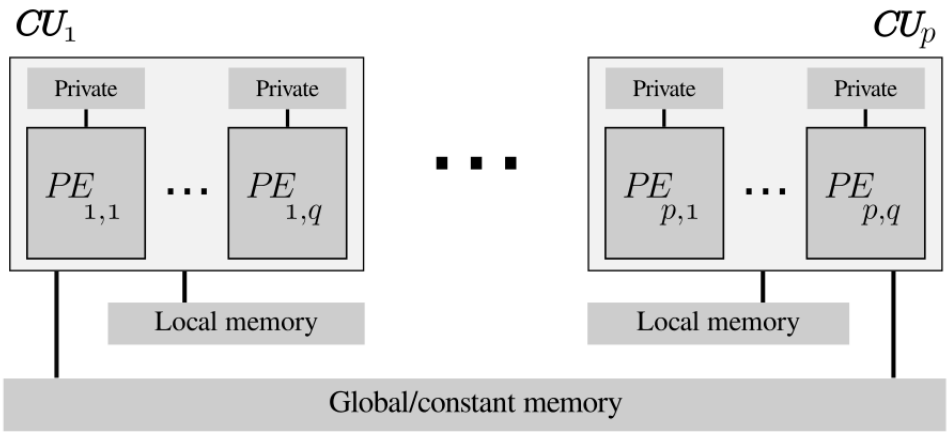


Figura 17: Conceito da arquitetura de dispositivos (AUGUSTO; BARBOSA, 2012).

A Figura 18 mostra como o OpenCL realiza a paralelização na GPU, com a atribuição de valores para X e Y, então a multiplicação de $X * Y$ determina a quantidade de *work-itens* (processos ou *threads*) que irão existir na placa. Ao agrupar uma quantidade p de *work-itens*, determina-se um *work-group*, que é executado utilizando a arquitetura SIMD. Essa divisão pode ser realizada entre uma e três dimensões, sendo obrigatório que o resultado da multiplicação da quantidade de *work-groups* de cada dimensão (equação 8), não deve ultrapassar a quantidade de *work-groups* máxima da placa (equação 9), sendo que essa quantidade é fornecida pela própria placa em uso.

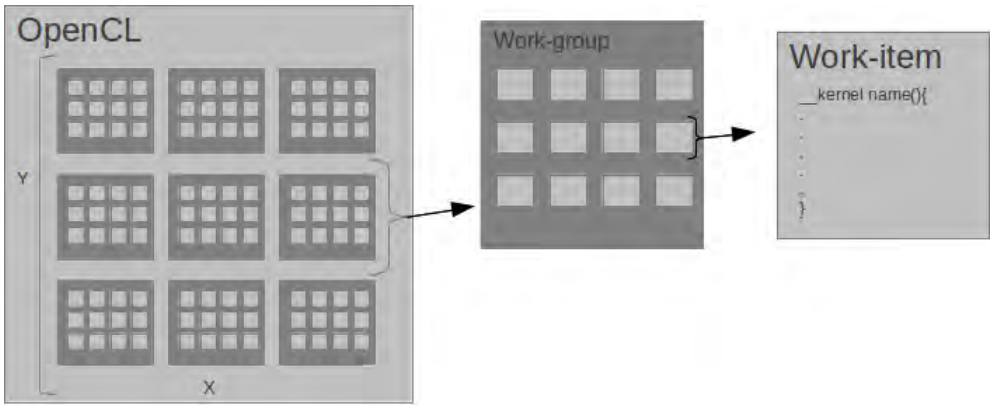


Figura 18: *Work-itens* e *work-group* do OpenCL.

$$WG_t = D_1 * D_2 * D_3 \tag{8}$$

Onde, WG_t representa a quantidade de *work-groups* total e D representa a quantidade de *work-groups* daquela dimensão.

$$WG_t <= WG_T \tag{9}$$

Onde WG_T representa a quantidade de *work-groups* suportada pela placa de vídeo. Outra obrigatoriedade é que a quantidade de *work-itens* de cada dimensão deve ser múltipla da quantidade de *work-group* da dimensão em questão (equação 10), ou seja, a divisão deve resultar em um número inteiro $\mathbb{Z}$.

$$\mathbb{Z} = \frac{WI_i}{WG_i} \tag{10}$$

Onde WI_i representa a quantidade de *work-itens* da dimensão $i(i = 1..3)$. Com essa estrutura pode-se dividir tarefas em tarefas menores, que são executadas simultaneamente em *work-itens* diferentes, gerando ganho de desempenho para resolução do problema aplicado.

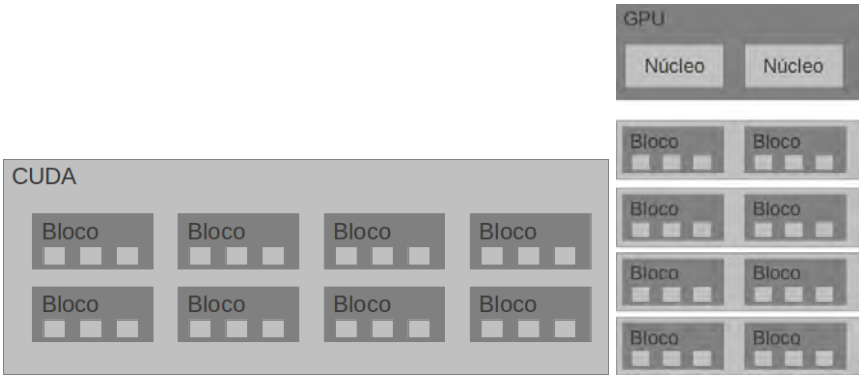
Outra característica do OpenCL é a existência de memórias com restrições e acessos diferentes. Pode-se dividir as memórias em 4 (NVIDIA, 2009b), como pode-se visualizar na Figura 17:

- Global: todos os *work-itens* possuem acesso para leitura e escrita.
- Local: somente os *work-itens* de um *work-group* pode ter acesso de leitura e escrita.
- Constante: todos os *work-itens* possuem acesso para leitura.
- Privado: cada *work-item* possui essa memória para escrita e leitura.

CUDA

CUDA é uma arquitetura e um modelo de programação paralela em placas GPU NVIDIA, para resolver problemas computacionais complexos de uma forma mais eficiente do que em uma CPU. Foi desenvolvida também para suportar várias linguagens como o CUDA C, CUDA Fortran e OpenCL (NVIDIA, 2011).

O CUDA funciona agrupando *threads* criadas pelo programa em blocos (Figura 19(a)), e cada bloco é então executado independentemente do outro em um determinado núcleo (NVIDIA, 2011), como mostra a Figura 19(b). Os blocos também são agrupados em *grids*.



(a) (b)

Figura 19: (a) Divisão em blocos das *threads* em CUDA; (b) Divisão dos blocos nos núcleos em CUDA

No CUDA existe algumas restrições como acontece com o OpenCL. Uma delas é que o total da multiplicação da quantidade de *threads* por blocos de cada dimensão, não pode ultrapassar a quantidade total de *threads* por blocos suportadas pela placa (equação 11).

$$T_T >= T_1 * T_2 * T_3 \tag{11}$$

Onde T_T é a quantidade de total de *threads* por bloco suportada pela placa e T_i (i = 1..3), é a quantidade de *threads* por bloco da dimensão i. Outra restrição é que a quantidade de blocos por *grid* por dimensão não pode ultrapassar a quantidade de blocos por *grid* dessa dimensão suportada pela placa (equação 12).

$$G_i >= g_i \tag{12}$$

Onde G_i representa a quantidade de blocos por *grid* da dimensão i (i = 1..3) suportada pela placa, e g_i representa a quantidade de blocos por *grid* da dimensão i definida no programa desenvolvido.

O CUDA divide a memória em 5 tipos (NVIDIA, 2011):

- Global: todas as *threads* tem acesso de escrita e leitura.
- Constante: todas as *threads* tem acesso de leitura.
- Textura: todas as *threads* tem acesso de leitura, especializado para tratamento de imagens.

- Compartilhada: somente as *threads* de um determinado bloco tem acesso de leitura e escrita.
- Privado: cada *thread* possui um espaço para variáveis privadas.

A Figura 20 mostra o esquema da hierarquia das memórias no CUDA, em que cada *thread* possui sua memória privada (P), as *threads* de cada bloco acessam a memória compartilhada e todos acessam as memórias global, constante e textura.

Figura 20: Hierarquia das memórias no CUDA

3 *Materiais e Métodos*

Para atingir o objetivo deste trabalho que é facilitar o processamento de dados ambientais com a utilização de programação paralela através das GPU's, com as linguagem CUDA e OpenCL, foi desenvolvido uma biblioteca que encapsule as configurações e os tratamentos de erros necessários para cada linguagem.

O desenvolvimento foi realizado na linguagem Java utilizando a biblioteca JOCL (JOCL, 2012) para OpenCL e JCUDA (JCUDA, 2012) para o CUDA. Essas bibliotecas foram utilizadas para realizar a conexão com a GPU, enviando e recebendo as respostas das respectivas linguagens. O encapsulamento consistiu em ocultar algumas ações necessárias para a utilização da GPU em cada linguagem, a fim de facilitar e otimizar o tempo de desenvolvimento para a programação paralela.

3.1 Execução do OpenCL e CUDA

Tanto o OpenCL quanto o CUDA possuem a seguinte sequência de execução que está representado na Figura 21, sendo que cada linguagem possui funções diferentes e tratamentos de erros diferentes para cada etapa. A execução no OpenCL e CUDA acontece da seguinte forma (GROUP, 2011; NVIDIA, 2011) (figura 21):

- Etapa 1: Consiste em descobrir quantos dispositivos paralelos estão disponíveis.
- Etapa 2: Identifica quais são os dispositivos para poder obter informações como quantidade de núcleos que possui, memória entre outras especificações.
- Etapa 3: Para cada dispositivo disponível é criado um contexto. O contexto contém qual é o dispositivo que será usado pelo *host* (máquina hospedeira), as funções que serão executadas neste contexto, o código fonte e o código executável, além dos objetos em memória que serão utilizados pelo contexto.
- Etapa 4: Cria-se a fila de comandos que serão executados no dispositivo determinado.

- Etapa 5: Compila o código fonte para o dispositivo selecionado, isso acontece porque cada dispositivo pode possuir funções diferentes para operações iguais.
- Etapa 6: Define-se o *kernel* (OpenCL) ou função (CUDA) que será executada.
- Etapa 7: Os espaços em memória para as variáveis, passadas por parâmetro do *host* para o dispositivo são criados.
- Etapa 8: Os dados dos parâmetros são copiados do *host* para o dispositivo.
- Etapa 9: A quantidade de *work-groups* e *work-itens* para o OpenCL, e de *grids* e *threads* por bloco para o CUDA, são determinadas para cada dimensão.
- Etapa 10: É executado o *kernel* ou função no dispositivo em questão.
- Etapa 11: Os dados de resposta são enviados do dispositivo para o *host*.
- Etapa 12: Finaliza-se a execução do dispositivo limpando as variáveis, a fila de comando e o contexto para o dispositivo.

Pode-se observar que as etapas 3 até 12 são executadas para cada dispositivo disponível e selecionado para a execução.

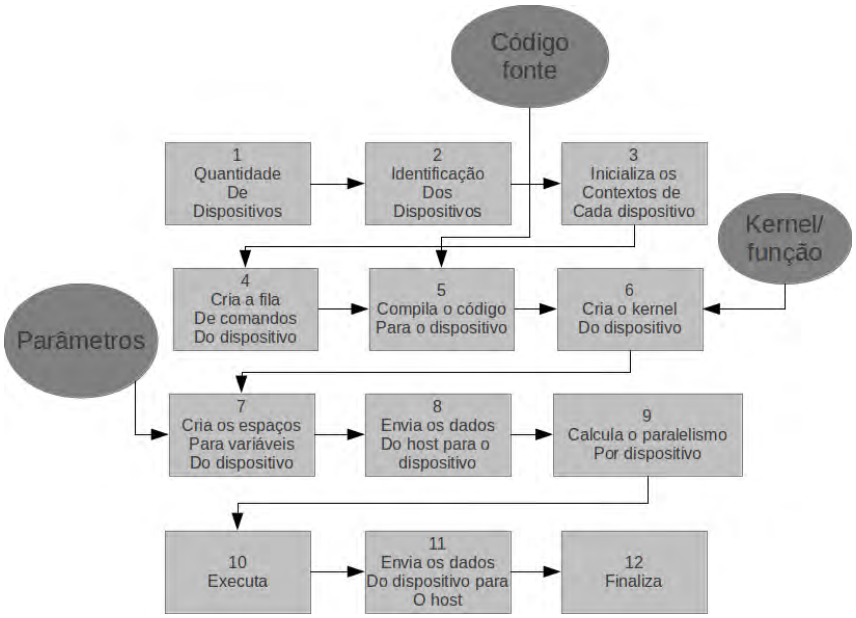


Figura 21: Sequência de execução do OpenCL/CUDA

3.2 Método Aplicado

O encapsulamento foi realizado agrupando todas as etapas de 1 até 12, da Figura 21, em uma única etapa, para ambas as linguagens. Esse encapsulamento foi implementado e denominado de GenericOpenCL e GenericCUDA.

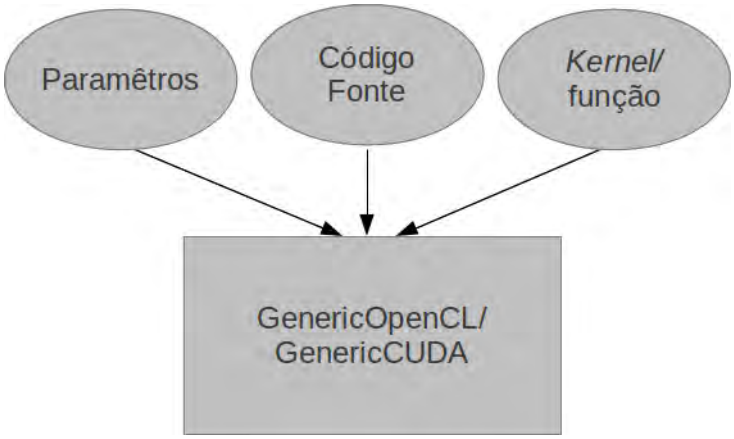


Figura 22: Encapsulamento da execução do OpenCL/CUDA

A Figura 22 mostra como ficou a sequência de execução após realizar o encapsulamento. Em ambas as linguagens é necessário enviar somente 3 parâmetros de entrada, tanto para o GenericOpenCL e quanto para o GenericCUDA. O primeiro é os dados, o segundo é o código-fonte da respectiva linguagem e o terceiro é o *kernel* para o OpenCL e o método para o CUDA, a ser executado na GPU.

O segundo parâmetro é o código-fonte, que representa o código que será executado na GPU, por exemplo, se é necessário realizar a soma entre dois vetores, pode-se enviar o seguinte código-fonte em CUDA (Algoritmo 3), para a biblioteca desenvolvida, escolhendo como terceiro parâmetro a função 'somando':

Algoritmo 3 Soma de dois vetores em CUDA

```
1: __global__ somando(int a, int b, int c) {
2: int index = threadIdx.x;
3: c[i] = a[i] + b[i];
4: }
```

Foi desenvolvida uma classe denominada Parâmetro, que tem por objetivo determinar as configurações de cada parâmetro passado para a biblioteca. Essa classe possui sua estrutura como mostra a Figura 23.

Os atributos dados, dadosFloat, dadosInt, dadosLong determinam qual tipo de dados será enviado para a GPU. O atributo *read* determina que o parâmetro será de leitura e o atributo *write*

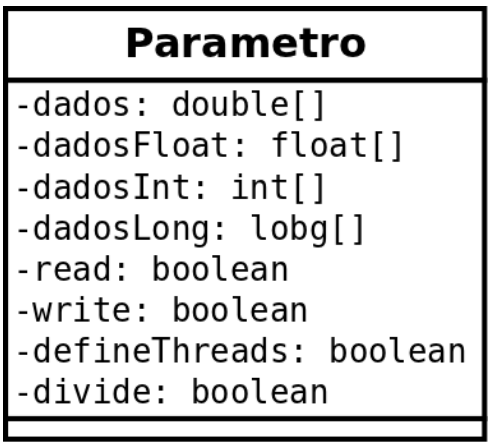


Figura 23: Diagrama UML da classe Parâmetro

determina se será de escrita.

Já o atributo *divide* determina se o parâmetro será divido entre os dispositivos existentes ou se será copiado integralmente para cada dispositivo, ou seja, caso tenha 2 dispositivos disponíveis e a quantidade de dados do parâmetro é de 10000, se o atributo *divide* foi configurado como verdadeiro cada dispositivo irá receber 5000 dados, portanto cada dispositivo irá precisar de menos memória para armazenar os dados. Esse atributo foi criado, pois existem cálculos onde não é necessário utilizar todo o conjunto de dados para cada iteração, porém existem outros onde todo o conjunto é utilizado em cada iteração, como ocorre com a dimensão fractal.

A biblioteca considera que o paralelismo será definido pela quantidade de dados enviados para ela, por isso existe um atributo denominado *defineThreads*, que tem a função de determinar se esse conjunto de dado, irá definir uma dimensão para a GPU. Essa última configuração é utilizada também para realizar os cálculos de quantas dimensões irão existir, quantos *work-groups* e *work-itens* por dimensão irão existir no OpenCL, por exemplo, considerando a seguinte configuração para o OpenCL:

1. A quantidade de dados do parâmetro é 1024;
2. O atributo *defineThreads* foi definido como verdadeiro;
3. A quantidade de *work-groups* suportada pela placa é de 256.

Logo para o OpenCL tem-se que a quantidade de *work-groups* para essa execução deverá ser de 16 e a quantidade de *work-itens* deverá ser 64 em uma única dimensão, sendo que todos esses valores foram calculados automaticamente pela biblioteca

O atributo *defineThreads* determina também quantas *grids*, blocos e *threads* por bloco irão existir no CUDA, por exemplo, considerando a seguinte configuração:

1. A quantidade de dados do parâmetro é 1024;
2. O atributo *defineThreads* foi definido como verdadeiro;
3. A quantidade de blocos por *grid* é de 2048;
4. A quantidade de *threads* por bloco é de 512.

Logo para o CUDA a quantidade de blocos por *grid* deverá ser 2 e a quantidade de *threads* por bloco de 512, utilizando somente uma dimensão.

3.3 Estudo de Caso

Após a implementação das bibliotecas foi realizado um estudo de caso para verificação da utilização das mesmas. O estudo baseou-se em 100.000 mil dados gerados da equação de Lorenz para o eixo X, pois conhece-se os valores de entrada e saída para a validação dos algoritmos desenvolvidos. Para esse estudo foram implementados 4 aplicativos listados a seguir:

1. Sequencial: Implementou-se o cálculo da dimensão fractal de forma sequencial. Após a implementação utilizou-se os dados da equação de Lorenz para validar os aplicativos.
2. OpenCL: O cálculo foi implementado na linguagem OpenCL, utilizando a biblioteca GenericOpenCL.
3. CUDA: Implementou-se a dimensão fractal em CUDA, utilizando a biblioteca GenericCUDA.
4. Teste: Esse aplicativo foi desenvolvido com o intuito de automatizar todos os testes. O aplicativo consistiu em 5 classes, onde 3 classes foram voltadas para utilizar os aplicativos anteriores, passando algumas configurações como os raios, as dimensões e quantas vezes cada teste seria executado. Uma classe foi desenvolvida para realizar a verificação dos resultados gerados, ou seja, para cada teste os dados foram comparados entre si para verificar quantas casas decimais tinham de diferença. E a última classe foi implementada para calcular as médias dos resultados para posterior plotagem em gráfico.

Após realizar todos os testes calculou-se o *speedup* de cada teste para análise.

3.3.1 Modelando o problema

Para utilizar o paralelismo foi aplicado a metodologia PCAM de Foster sobre o cálculo da dimensão fractal. Sendo cada etapa descrita abaixo:

1. Decomposição: Foi decomposto a equação 3 em raios e dimensões escolhidos, dados e função *heaviside*.
2. Comunicação: Verificou-se que os raios e dimensões possuem pouca comunicação, como pode-se visualizar pela equação 4, pois é necessário realizar o cálculo somente com 1 raio e 1 dimensão, enquanto que os dados comunica-se intensamente entre si, pois para cada par de ponto é calculado a função *heaviside*.
3. Aglomeração: Aglomerou-se as partes dos pares de pontos para o cálculo do *heaviside*, para cada raio e dimensão.
4. Mapeamento: Mapeou-se as aglomerações entre os *work-groups* e *work-itens* do OpenCL e *threads* por bloco e blocos por *grid* para o CUDA.

3.4 Testes

Para a realização dos testes foi utilizado uma máquina com uma placa de vídeo NVIDIA GT 520m que continha 48 núcleos com velocidade de 1,5 GHz. Essa máquina possuía um processador I5 com 4 núcleos de 2,30 GHz cada, e 6 GB de RAM, com o Ubuntu 11.10 64 bits.

Foram realizados dois tipos de teste, sendo que todos os teste tiveram a quantidade de dimensões fixadas em 9 pois cada dimensão possui a mesma influência no cálculo que um raio:

1. Quantidade de Dados: Foi utilizado somente uma fração dos dados por teste, (executou-se os testes com 1.000, 2.000, 4.000, 8.000, 16.000, 32.000 e 64000 dados, tanto com OpenCL e CUDA, quanto com a programação sequencial), utilizou-se também todos os 100.000 dados para o OpenCL e CUDA, esse último não foi realizado no sequencial por falta de disponibilidade de tempo para uso do computador de teste. Neste teste a quantidade de raios foi fixada em 40. Essa variação de dados testou se a diferença de velocidade da GPU com relação a CPU poderia ter uma influência no comportamento do tempo de resposta.
2. Quantidade de Raios: Outra variação de teste, além da quantidade de dados, foi a variação na quantidade de raios para o cálculo, para testar o quanto a paralelização influenciou no

desempenho. A quantidade de raios variou entre 100 e 1000, tendo um acréscimo de 50 raios por teste, totalizando 19 pontos por teste. Neste teste a quantidade de dados foi fixada ora em 1000, 2000 ou 4000.

Os algoritmos utilizados para os testes podem ser encontrador a seguir 4, 5 e 6. Ao analisá-los percebe-se que os loops das linhas dois e três do algoritmo sequencial já não existem no algoritmo do OpenCL e no CUDA. Isso se deve a combinação dos valores desses dois loops para realizar a paralelização dentro da GPU através de um vetor resultante dos loops removidos. Essa combinação dos vetores e a leitura dos dados foram feitos em Java e passados como parâmetro para as bibliotecas criadas do OpenCL e CUDA. O tempo gasto para a execução de cada algoritmo foi medido e plotado em gráficos para análise.

Algoritmo 4 Algoritmo Sequencial

```
1:  inicializa dados
2:  for d = 2 → maxDim do
3:      for lr = minR → maxR do
4:          for i = 0 → tamanho_dos_dados − 1 do
5:              for j = i + 1 → tamanho_dos_dados do
6:                  for n = 0 → d do
7:                      soma as distâncias ao quadrado
8:                  end for
9:                  executa a função Heaviside()
10:             end for
11:         end for
12:     end for
13: end for
```

Algoritmo 5 Algoritmo Opencl

```
1:  inicializa dados
2:  d = get_global_id(0)
3:  for i = 0 → tamanho_dos_dados − 1 do
4:      for j = i + 1 → tamanho_dos_dados do
5:          for n = 0 → d do
6:              soma as distâncias ao quadrado
7:          end for
8:          executa a função Heaviside()
9:      end for
10: end for
```

Algoritmo 6 Algoritmo CUDA

```
1: inicializa dados
2: d = blockIdx.x * blockDim.x + threadIdx.x;
3: if d < total_paralelizado then
4:     for i = 0 → tamanho_dos_dados − 1 do
5:         for j = i + 1 → tamanho_dos_dados do
6:             for n = 0 → d do
7:                 soma as distâncias ao quadrado
8:             end for
9:             executa a função Heaviside()
10:        end for
11:    end for
12: end if
```

4 Resultado e Discussão

4.1 Comparação das bibliotecas desenvolvidas

A seguir é mostrado o algoritmo sem a utilização da biblioteca desenvolvida para a utilizar os recursos da GPU na linguagem CUDA.

Algoritmo 7 Algoritmo para utilizar o CUDA sem a biblioteca desenvolvida

```
1:  inicializa os dados
2:  JCudaDriver.setExceptionsEnabled(true);
3:  String ptxFileName = preparePtxFile(codigo);
4:  cuInit(0);
5:  CUdevice device = new CUdevice();
6:  cuDeviceGet(device, 0);
7:  CUcontext context = new CUcontext();
8:  cuCtxCreate(context, 0, device);
9:  CUmodule module = new CUmodule();
10: cuModuleLoad(module, ptxFileName);
11: CUfunction function = new CUfunction();
12: cuModuleGetFunction(function, module, metodo);
13: CUdeviceptr deviceInput = new CUdeviceptr();
14: Pointer point = Pointer.to(parametro.getDados());
15: cuMemAlloc(deviceInput, parametro.getSize() * Sizeof.DOUBLE);
16: cuMemcpyHtoD(deviceInput, aux, parametro.getSize() * Sizeof.DOUBLE);
17: Pointer kernelParameters = Pointer.to(point);
18: int blockSizeX = 2;
19: int blockSizeY = 2;
20: int blockSizeZ = 2;
21: int gridSizeX = 2;
22: int gridSizeY = 1;
23: int gridSizeZ = 1;
24: int threadsPerBlock = 64;
25: cuLaunchKernel(function,
26: gridSizeX, gridSizeY, gridSizeZ, // Blocos das Grids
27: blockSizeX, blockSizeY, blockSizeZ, // Threads dos blocos
28: 0, null,kernelParameters, null);
29: cuCtxSynchronize();
30: cuMemcpyDtoH(point, deviceInput, SIZE * Sizeof.DOUBLE);
31: cuMemFree(deviceInput);
```

Em seguida tem-se o algoritmo para o CUDA utilizando a biblioteca desenvolvida.

Algoritmo 8 Algoritmo para utilizar o CUDA com a biblioteca desenvolvida

```
1: inicializa os dados
2: List<Parametro> parametros = new ArrayList<Parametro>();
3: Parametro parametro = new Parametro();
4: parametro.setDados(dados);
5: parametro.setRead(true);
6: parametro.setWrite(true);
7: parametro.setDefineThreads(true);
8: parametros.add(parametro);
9: GenericCUDA.executar(parametros, codigo, kernel);
```

O algoritmo sem a utilização da biblioteca desenvolvida para a linguagem OpenCL é mostrado a seguir.

Algoritmo 9 Algoritmo para utilizar o OpenCL sem a biblioteca desenvolvida

```
1: inicializa os dados
2: Pointer pointer = Pointer.to(dados);
3: final int platformIndex = 0;
4: final long deviceType = CL_DEVICE_TYPE_GPU;
5: final int deviceIndex = 0;
6: CL.setExceptionsEnabled(true);
7: int numPlatformsArray[] = new int[1];
8: clGetPlatformIDs(0, null, numPlatformsArray);
9: int numPlatforms = numPlatformsArray[0];
10: cl_platform_id platforms[] = new cl_platform_id[numPlatforms];
11: clGetPlatformIDs(platforms.length, platforms, null);
12: cl_platform_id platform = platforms[platformIndex];
13: cl_context_properties contextProperties = new cl_context_properties();
14: contextProperties.addProperty(CL_CONTEXT_PLATFORM, platform);
15: int numDevicesArray[] = new int[1];
16: clGetDeviceIDs(platform, deviceType, 0, null, numDevicesArray);
17: int numDevices = numDevicesArray[0];
18: cl_device_id devices[] = new cl_device_id[numDevices];
19: clGetDeviceIDs(platform, deviceType, numDevices, devices, null);
20: cl_device_id device = devices[deviceIndex];
21: cl_context context = clCreateContext(contextProperties, 1, new cl_device_id[]device, null, null, null);
22: cl_command_queue commandQueue = clCreateCommandQueue(context, device, 0, null);
23: int[] error = null;
24: cl_program program = clCreateProgramWithSource(context, 1, new String[]codigoFonte, null, null);
25: clBuildProgram(program, 0, null, null, null, null);
26: cl_kernel kernel = clCreateKernel(program, metodo, error);
27: cl_mem memObjects[] = new cl_mem[1];
28: memObjects[0] = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, Sizeof.cl_double * SIZE, pointer, error);
29: clEnqueueWriteBuffer(commandQueue, memObjects[i], CL_TRUE, 0, Sizeof.cl_double * SIZE, pointer, 0, null, null);
30: clSetKernelArg(kernel, i, Sizeof.cl_mem, Pointer.to(memObjects[0]));
31: long global_work_size[] = new long[]{16};
32: long local_work_size[] = new long[]{2};
33: int dim = 1;
34: clEnqueueNDRangeKernel(commandQueue, kernel, dim, null, global_work_size, local_work_size, 0, null, null);
35: clFinish(commandQueue);
36: clEnqueueReadBuffer(commandQueue, memObjects[0], CL_TRUE, 0, SIZE * Sizeof.cl_double, pointer, 0, null, null);
37: clReleaseMemObject(memObjects[0]);
38: clReleaseKernel(kernel);
39: clReleaseProgram(program);
40: clReleaseCommandQueue(commandQueue);
41: clReleaseContext(context);
```

A seguir visualiza-se o algoritmo para o OpenCL utilizando a biblioteca desenvolvida.

Algoritmo 10 Algoritmo para utilizar o OpenCL com a biblioteca desenvolvida

```
1: inicializa os dados
2: List<Parametro> parametros = new ArrayList<Parametro>();
3: Parametro parametro = new Parametro();
4: parametro.setDados(dados);
5: parametro.setRead(true);
6: parametro.setWrite(true);
7: parametro.setDefineThreads(true);
8: parametros.add(parametro);
9: GenericOpenCL.executar(parametros, codigo, kernel);
```

Os algoritmos 7, 8, 9 e 10 utilizaram somente uma variável como parâmetro para a GPU. Nos algoritmos 7 e 9 quanto mais variáveis forem utilizadas na GPU maior será a complexi-

dade do algoritmo, além de, para cada dispositivo GPU as funções desses algoritmos devem ser ajustadas devido a capacidades diferentes. Os algoritmos 8 e 10 mostram como ficou fácil utilizar o processamento em GPU utilizando as bibliotecas desenvolvidas. Com isso o desenvolvedor/programador ganha agilidade para se preocupar com a solução do problema e não em saber como utilizar a GPU, além de obter portabilidade entre dispositivos e consequentemente uma diminuição na manutenção do código-fonte.

A tabela 1 mostra onde cada etapa da Figura 21 pode ser encontrada nos algoritmos 7 e 9. Os tratamentos de erros e outras funções foram omitidas por serem desnecessárias para a compreensão do código.

Tabela 1: Tabela que associa as etapas da Figura 21 com as linhas dos algoritmos 7 e 9

Etapas:	CUDA	OpenCL
Etapa 1: Quantidade de dispositivos	Omitido	7,8,9
Etapa 2: Identificação dos dispositivos	4,5,6	10 até 19
Etapa 3: Inicializa os contextos	7,8	20,21
Etapa 4: Cria as filas de execução	9,10	22
Etapa 5: Compila o código-fonte	3	24,25
Etapa 6: Cria o kernel/função	11,12	26
Etapa 7: Cria os espaços para as variáveis	13,14,15	27,28
Etapa 8: Envia os dados	16,17	29
Etapa 9: Calcula o paralelismo	18 até 24	31,32,33
Etapa 10:Executa	25,29	34
Etapa 11:Retorna os dados	30	36
Etapa 12:Finaliza	31	37 até 41

A Figura 24(a) mostra a arquitetura para utilizar a GPU sem as bibliotecas desenvolvidas. O desenvolvedor/programador inicia os dados na linguagem Java, então utiliza a biblioteca JCUDA ou JOCL realizar as configurações necessárias na GPU para executar o programa. Para posteriormente enviar o código CUDA ou OpenCL para a GPU determinando sua execução. Já a Figura 24(b) mostra como ficou a arquitetura para utilizar a GPU com as bibliotecas desenvolvidas. Com o uso da solução proposta, o desenvolvedor/programador simplesmente inicia os dados na linguagem Java, configurando cada conjunto de dados pela classe Parametro, determinando o que cada um será na GPU, então o desenvolvedor/programador envia os parâmetros junto com o código-fonte do CUDA ou do OpenCL para as respectivas bibliotecas GenericCUDA e GenericOpenCL que fica responsável por configurar, enviar os dados, executar,

recuperar os dados, tratar os erros e finalizar a execução na GPU.

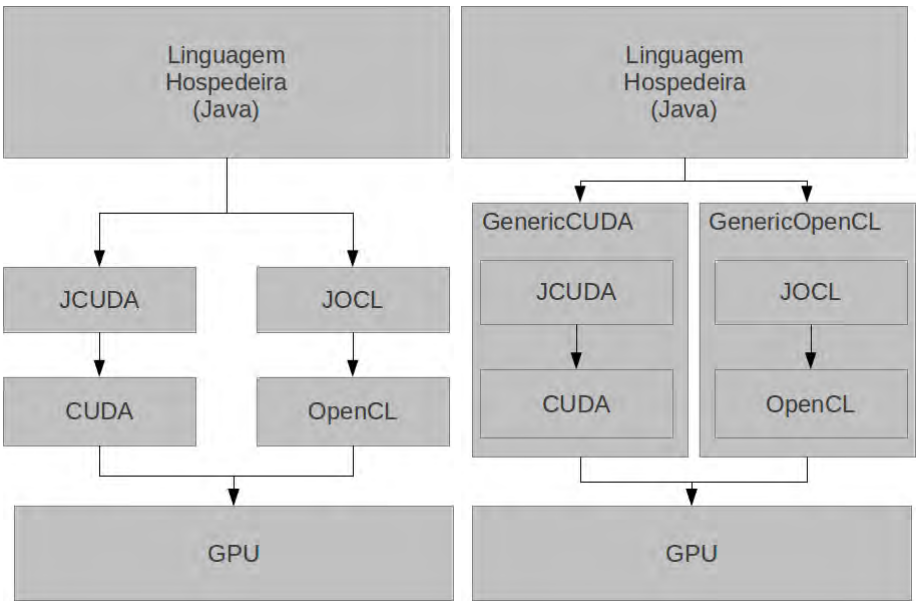


Figura 24: (a) Arquitetura de como utilizar a GPU sem as bibliotecas desenvolvidas. (b) Arquitetura de como utilizar a GPU com as bibliotecas desenvolvidas.

4.2 Testes 1 - Quantidade de Dados

4.2.1 Tempo

As Figuras 25(a) e 25(b) ilustram o comportamento da medida de tempo entre os algoritmos OpenCL, CUDA e o sequencial para o teste da variação na quantidade de dados.

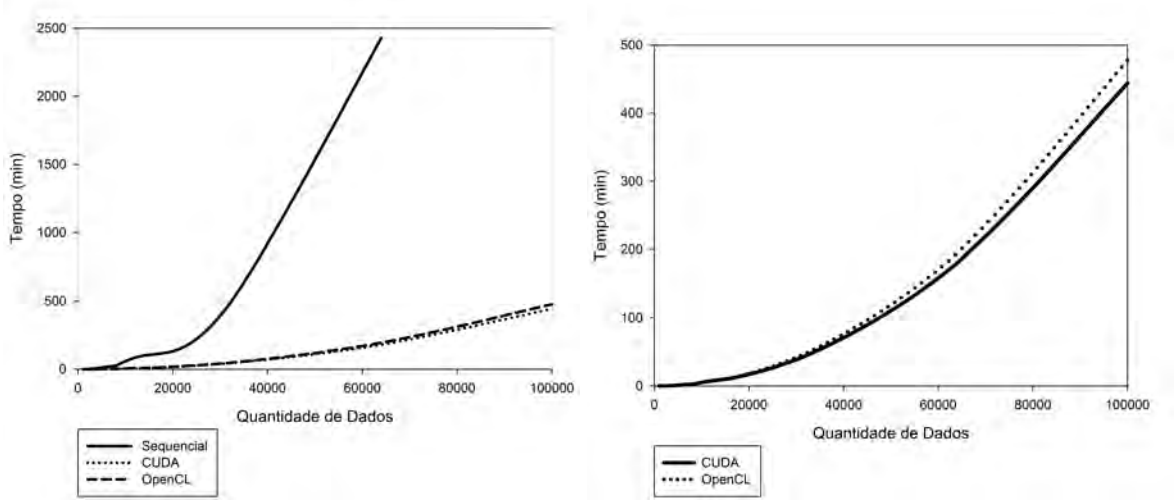


Figura 25: (a) Gráfico representa o comportamento do tempo em relação a quantidade de dados. (b) Gráfico representa um *zoom* do (a), utilizando somente os resultados do CUDA e OpenCL.

Ao observar as Figuras 25(a) e 25(b) pode-se concluir que a utilização do OpenCL e do CUDA para o cálculo da dimensão fractal reduziu significativamente o tempo, realizando uma redução de 89% à 91% para o OpenCL e de 87% até 92% para o CUDA. Isso se deve ao fato da fórmula permitir sua paralelização, pois pode-se calcular raios diferentes (Ri, ..., Rn) junto com dimensões diferentes (Di, ..., Dn) em paralelo dependendo da quantidade de processos simultâneos permitido pela placa GPU. Assim, quanto mais núcleos a placa GPU tiver, mais *work-groups* ou *grids* ela poderá ter e consequentemente mais distâncias/raios poderão ser paralelizadas.

A Figura 25(a) nos permite visualizar que mesmo utilizando a mesma ideia para os algoritmos, o OpenCL e o CUDA têm um aproveitamento cada vez melhor quando se aumenta a quantidade de dados, mesmo considerando que a velocidade da GPU (1,5 GHz) é menor que do CPU (2,3 Ghz). Isso talvez aconteça porque a troca entre as *threads* dentro de GPU é mais rápida pelo fato de serem *lightweight threads*, ou seja, utilizam recursos (memória) semelhantes, portanto ocupam menos espaço, consequentemente sendo mais rápido a troca entre elas.

4.2.2 *Speedup*

A Figura 26(a) ilustra o comportamento do *speedup* dos algoritmos em OpenCL e CUDA em relação ao algoritmo sequencial para o teste da variação na quantidade de dados. A Figura 26(b) mostra o *speedup* do CUDA em relação ao OpenCL.

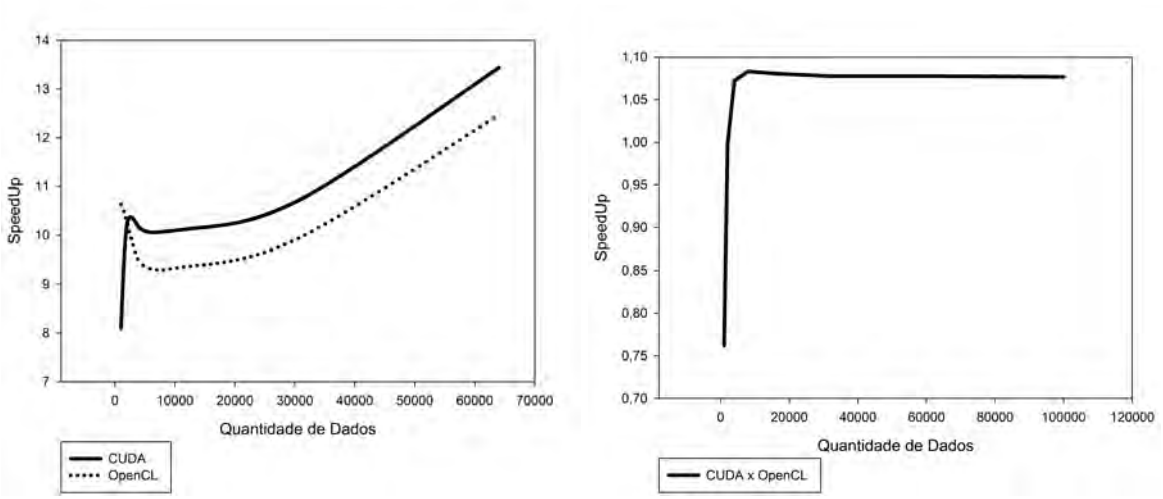


Figura 26: (a) Gráfico representa o comportamento do *speedup* do CUDA e OpenCL, quando variou-se a quantidade de dados. (b) Gráfico representa o *speedup* do CUDA em relação ao OpenCL.

Observa-se pelas Figuras 26(a) e 26(b) que o CUDA possui um *speedup* inicial pior em relação ao OpenCL, 8,09 para o CUDA e 10,62 para o OpenCL quando tem-se 1000 dados, contudo após passar de 4000 dados o CUDA ultrapassa o OpenCL e a partir desse momento aparenta-se uma diferença contínua entre as duas linguagens, podendo observar essa tendência pela Figura 26(b), onde o *speedup* do CUDA em relação ao OpenCL estabiliza-se em 1,07, ou seja, para quantidade de dados maiores o CUDA obteve um ganho maior que o OpenCL.

A Tabela 2 mostra algumas informações sobre os resultados do teste da quantidade de dados, pode-se verificar que apesar do CUDA ter obtido uma média melhor, teve também um desvio padrão superior, fazendo com que a diferença entre o mínimo e o máximo fosse de 5,43, sendo esse valor superior ao encontrado no OpenCL que teve um amplitude de 3,18. Ao analisar a tabela 2, juntamente com as Figuras 26(a) e 26(b), conclui-se que o CUDA tem um desempenho menor quando existe poucos dados, isso pode ocorrer devido ao fato do CUDA precisar iniciar mais configurações que o OpenCL.

Tabela 2: Resultados dos *speedups* do teste de quantidade de dados

	Média	Desvio Padrão	Máximo	Mínimo
Cuda	10,42	1,58	13,44	8,10
OpenCL	10,22	1,10	12,47	9,29
CUDA x OpenCL	1,03	0,11	1,08	0,76

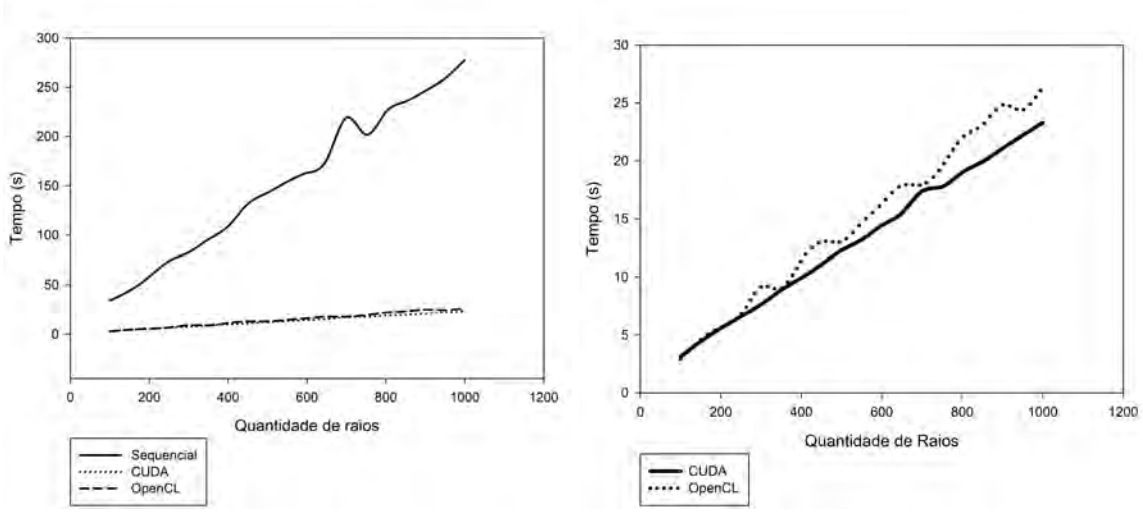
4.3 Testes 2 - Quantidade de Raio

4.3.1 Teste com 1000 Dados fixos

Neste teste a quantidade de dados foram fixadas em 1000 e variando a quantidade de raios.

Tempo

As Figuras 27(a) e 27(b) mostram o tempo de execução de cada teste. Pode-se observar que o CUDA obteve uma redução de tempo entre 90% até 92%, já o OpenCL teve uma redução de 88% até 92%.

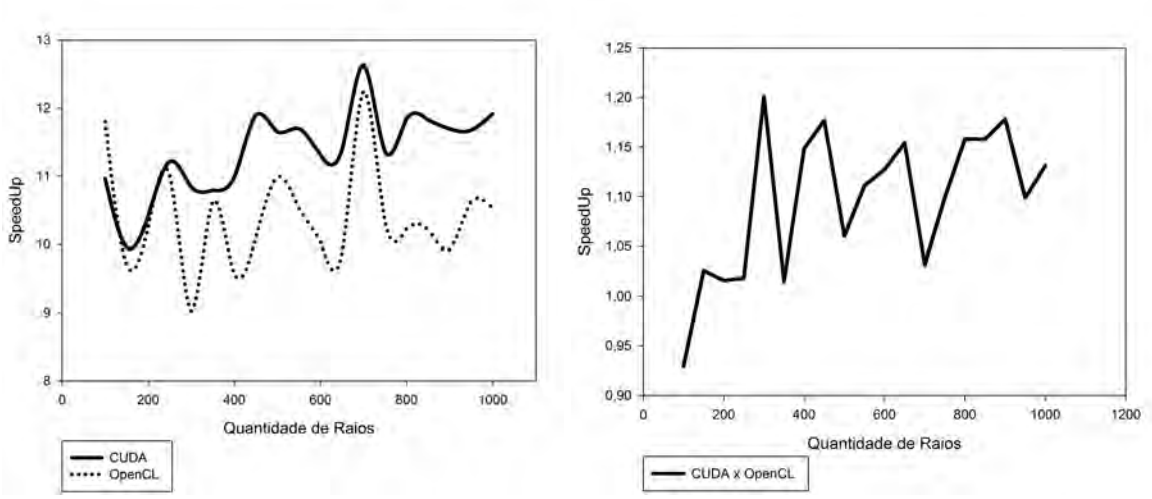


(a) (b)

Figura 27: (a) Gráfico representa o comportamento do tempo de resposta quando variou-se a quantidade de raios, fixando em 1000 a quantidade de dados. (b) Gráfico representa um *zoom* do (a), utilizando somente os resultados do CUDA e OpenCL.

Speedup

A Figura 28(a) mostra o comportamento do *speedup* para o CUDA e o OpenCL, observa-se que inicialmente o OpenCL apresenta um *speedup* melhor 11,8 de ganho contra 11 do CUDA. Contudo o CUDA torna-se superior possuindo uma média de 11,4 vezes mais rápido que o sequencial, enquanto o OpenCL apresenta 10,5 de média. O melhor desempenho do CUDA pode ser observado também na Figura 28(b), onde a média de *speedup* do CUDA com relação ao OpenCL foi de 1,09, ou seja, uma diferença de 9% entre eles.



(a) (b)

Figura 28: (a) Gráfico representa o comportamento do *speedup* do CUDA e OpenCL, quando variou-se a quantidade de raios, fixando em 1000 a quantidade de dados. (b) Gráfico representa o *speedup* do CUDA em relação ao OpenCL.

A Tabela 3 mostra os resultados do *speedup* do teste com 1000 dados fixos, observa-se uma variação menor que o primeiro teste realizado, sendo observado principalmente pelo desvio padrão pequeno, consequentemente obtendo uma amplitude pequena entre o mínimo e o máximo. Apesar disso houve alguns casos em que a diferença entre o CUDA e o OpenCL foi grande, atingindo até 1,20 de diferença, ou seja 20%.

Tabela 3: Resultados dos *speedups* dos testes com 1000 dados fixos

	Média	Desvio Padrão	Máximo	Mínimo
Cuda	11,37	0,62	12,64	9,96
OpenCL	10,40	0,75	12,25	9,02
CUDA x OpenCL	1,10	0,07	1,20	0,93

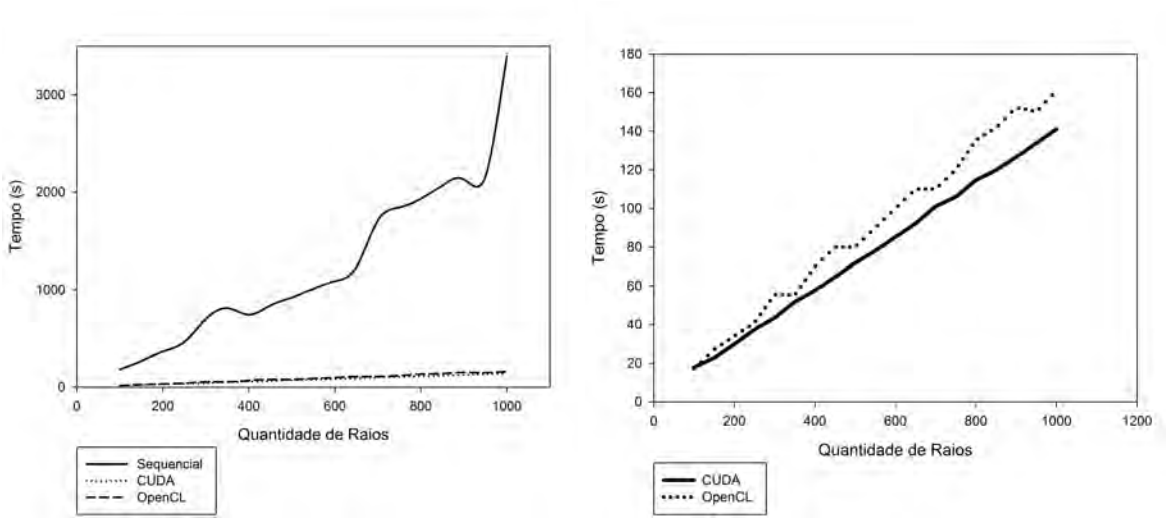
4.3.2 Teste com 2000 Dados fixos

Neste teste a quantidade de dados foram fixadas em 2000 e variou-se a quantidade de raios.

Tempo

As Figuras 29(a) e 29(b) mostram o tempo de execução com 2000 dados fixos. Foi observado que o CUDA obteve uma redução de tempo entre 90% até 96%, já o OpenCL teve uma

redução de 90% até 95%. Com relação ao teste mostrado na seção 4.3.1 aqui obteve-se uma redução ainda mais expressiva, pois uma característica importante para utilizar as GPU's é que a quantidade de cálculo por núcleo não deve ser tão pequena e nem tão grande para se obter um maior ganho. Pode-se verificar isso com os resultados desses dois testes.

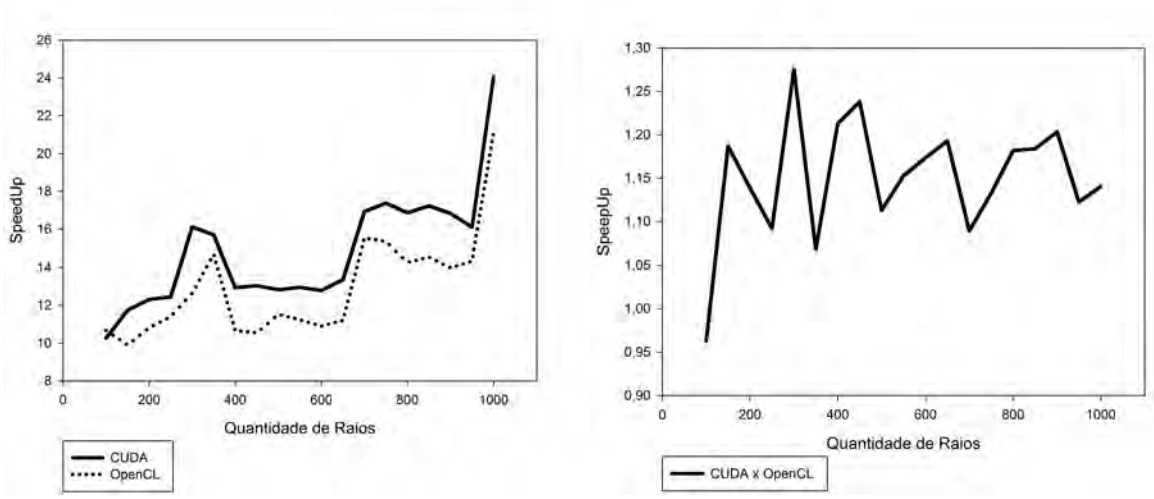


(a) (b)

Figura 29: (a) Gráfico representa o comportamento do tempo de resposta quando variou-se a quantidade de raios, fixando em 2000 a quantidade de dados. (b) Gráfico representa um *zoom* do (a), utilizando somente os resultados do CUDA e OpenCL.

Speedup

O comportamento do *speedup* pode ser observado na Figura 30(a), que mostra um *speedup* médio de 14,83 para o CUDA, e de 12,9 de média para o OpenCL. Novamente neste teste observa-se um melhor desempenho do CUDA, podendo ser observado também na Figura 30(b). A média de *speedup* do CUDA com relação ao OpenCL foi de 1,15, ou seja, uma diferença de 15% no tempo de execução.



(a) (b)

Figura 30: (a) Gráfico representa o comportamento do *speedup* do CUDA e OpenCL, quando variou-se a quantidade de raios, fixando em 2000 a quantidade de dados. (b) Gráfico representa o *speedup* do CUDA em relação ao OpenCL.

A Tabela 4 mostra os resultados do *speedup* do teste com 2000 dados fixos, nesse teste consegue-se observar uma grande variação entre os testes, atingindo um desvio padrão de 3,13 e uma amplitude de 13,79 para o CUDA e 11,20 para o OpenCL, o que representa uma variação considerável. Neste teste o CUDA começa a se distanciar em relação ao OpenCL no quesito *speedup*.

Tabela 4: Resultados dos *speedups* dos testes com 2000 dados fixos

	Média	Desvio Padrão	Máximo	Mínimo
Cuda	14,84	3,13	24,06	10,27
OpenCL	12,91	2,73	21,10	9,90
CUDA x OpenCL	1,15	0,07	1,28	0,96

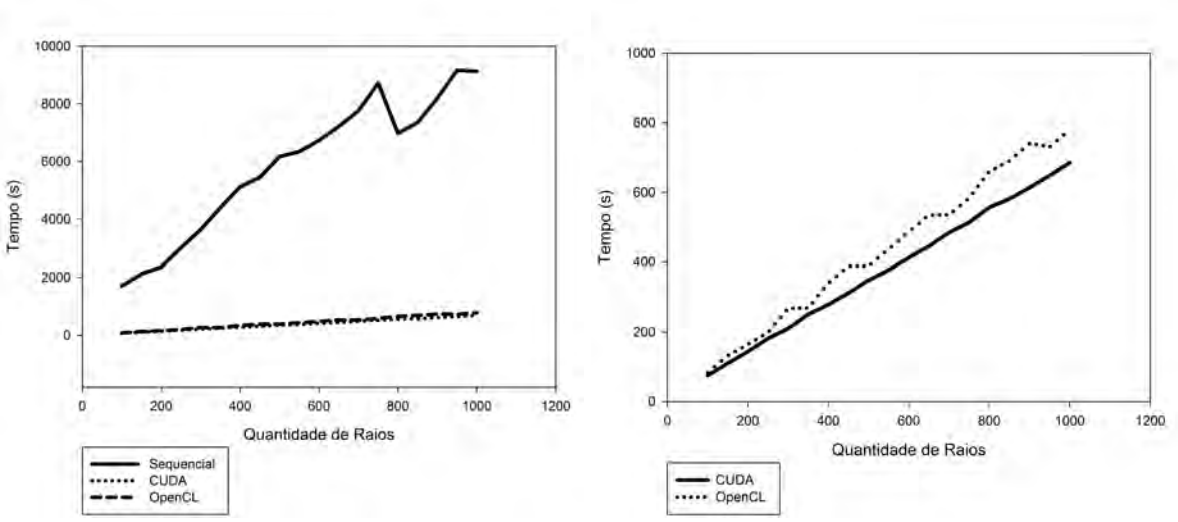
4.3.3 Teste com 4000 Dados fixos

Neste teste a quantidade de dados foram fixadas em 4000 e variando a quantidade de raios.

Tempo

As Figuras 31(a) e 31(b) mostram o tempo de execução com 4000 dados fixos. Foi observado que o CUDA obteve uma redução de tempo entre 92% até 96%, já o OpenCL teve uma

redução de 90% até 95%. Com relação aos testes mostrados na seção 4.3.1 obteve-se uma redução mais expressiva, mas com relação ao teste da seção 4.3.2 já não obteve uma melhora mais significativa.

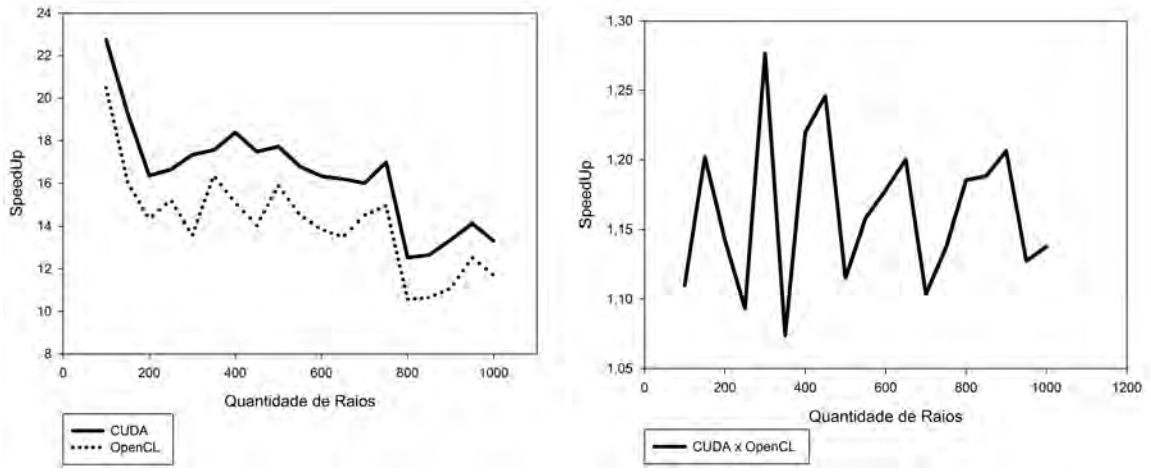


(a) (b)

Figura 31: (a) Gráfico representa o comportamento do tempo de resposta quando variou-se a quantidade de raios, fixando em 4000 a quantidade de dados. (b) Gráfico representa um *zoom* do (a), utilizando somente os resultados do CUDA e OpenCL.

Speedup

Na Figura 32(a) tem-se o gráfico da variação do *speedup*, onde o CUDA apresentou uma média de 16,4 e o OpenCL 14,1. Apesar de ter uma média melhor que a apresentada na seção 4.3.2, observa-se pelo comportamento do gráfico 32(a) que a tendência é de um *speedup* cada vez menor, enquanto no gráfico 30(a) a tendência é de crescimento do *speedup*.



(a) (b)

Figura 32: (a) Gráfico representa o comportamento do *speedup* do CUDA e OpenCL, quando variou-se a quantidade de raios, fixando em 4000 a quantidade de dados. (b) Gráfico representa o *speedup* do CUDA em relação ao OpenCL.

Pode-se observar em todas as Figuras das seções 4.2 e 4.3.1 que o CUDA apresenta um desempenho melhor que o OpenCL, isso acontece pelo fato de o CUDA ser projetado exclusivamente para GPU's, possuindo assim instruções específicas para aumentar o desempenho nas GPU's NVIDIA, enquanto o OpenCL é projetado para programação paralela para dispositivos que possuam paralelismo como GPU e CPU.

Através das Figuras 27(b), 29(b) e 31(b) observa-se comportamentos muito semelhantes, pois para cada raio incrementado o impacto na fórmula é o mesmo para todos os pontos, pois todos os pontos deverão passar por esse novo raio.

A Tabela 5 mostra os resultados do *speedup* do teste com 4000 dados fixos, nesse teste o desvio padrão começa a diminuir juntamente com o máximo atingido por ambas as linguagens, isso indica que existe um ponto ótimo para obter o maior ganho, que a GPU utilizada pode fornecer para o cálculo da dimensão fractal. Contudo mesmo diminuindo o ganho esse valor ainda representa uma grande redução no tempo de resposta independentemente de estar no ponto ótimo ou não.

Tabela 5: Resultados dos *speedups* dos testes com 4000 dados fixos

	Média	Desvio Padrão	Máximo	Mínimo
Cuda	16,41	2,49	22,75	12,54
OpenCL	14,15	2,34	20,50	10,58
CUDA x OpenCL	1,16	0,05	1,28	1,07

As tabelas 6, 7 e 8 mostram um resumo dos resultados dos *speedups* de todos os testes. Observa-se pela Tabela 8 que o CUDA vai melhorando com relação ao OpenCL e isso fica bem claro ao observar o comportamento do valor mínimo do *speedup* entre o CUDA e o OpenCL.

Tabela 6: Resumo dos resultados dos *speedups* dos testes para o CUDA

Testes	Média	Desvio Padrão	Máximo	Mínimo
Dados	10,42	1,58	13,44	8,10
1000	11,37	0,62	12,64	9,96
2000	14,84	3,13	24,06	10,27
4000	16,41	2,49	22,75	12,54

Tabela 7: Resumo dos resultados dos *speedups* dos testes para o OpenCL

Testes	Média	Desvio Padrão	Máximo	Mínimo
Dados	10,22	1,10	12,47	9,29
1000	10,40	0,75	12,25	9,02
2000	12,91	2,73	21,10	9,90
4000	14,15	2,34	20,50	10,58

Tabela 8: Resumo dos resultados dos *speedups* dos testes para o CUDAxOpenCL

Testes	Média	Desvio Padrão	Máximo	Mínimo
Dados	1,03	0,11	1,08	0,76
1000	1,10	0,07	1,20	0,93
2000	1,15	0,07	1,28	0,96
4000	1,16	0,05	1,28	1,07

5 *Conclusão*

O estudo de fenômenos naturais através da análise do comportamento de variáveis micrometeorológicas utiliza diversas ferramentas matemáticas para tentar compreender e explicar tais fenômenos. Uma das abordagens existentes é Teoria da Complexidade, cujo portfólio envolve diversos cálculos como a dimensão fractal, o expoente de Lyapunov, a análise de recorrência, entre outros. Neste trabalho, para manipulação das variáveis micrometeorológicas, foi escolhida a dimensão fractal, que pode ser utilizada para identificar o comportamento dos atratores e a independência das variáveis.

Para realização desses cálculos o custo computacional é muito alto requerendo o uso de arquiteturas computacionais de processamento paralelo como GPU, cluster e grid. Este trabalho buscou trabalhar com a solução de menor custo e desenvolveu uma solução para minimizar o tempo de resposta para cálculos utilizando a arquitetura de GPU's. Para isso foi desenvolvida uma biblioteca que torna transparente ao programador as configurações internas dessa arquitetura, ou seja, a biblioteca tem o intuito de facilitar o desenvolvimento de aplicações paralelas, encapsulando algumas configurações e alguns tratamentos de erros de forma automática. Foi desenvolvida duas bibliotecas para a utilização da programação paralela em GPU's, com as linguagens OpenCL e CUDA. Para validação do trabalho, foram realizados cálculos sobre séries temporais de dados ambientais.

Todos esses aspectos facilitam o desenvolvimento de cálculos utilizando programação paralela para dados ambientais. Assim pode-se ter uma diminuição do custo para os cálculos sobre os dados com utilização do paralelismo sem ser através de *clusters* que possuem maior custo.

Após a utilização das bibliotecas desenvolvidas nos testes pode-se observar nos resultados que a aplicação do OpenCL e CUDA para o cálculo de dimensão fractal mostrou-se muito eficiente. O método aplicado mostrou redução de 88% até 95% do tempo para a obtenção dos resultados utilizando a biblioteca em OpenCL e de 87% até 96% de redução utilizando a biblioteca em CUDA. Assim, a compreensão e estudos de novos fenômenos através da Teoria da Complexidade poderá ser realizada de forma mais rápida, possibilitando respostas ágeis para

o mundo científico.

Pode-se concluir também que a utilização das bibliotecas desenvolvidas agilizou e facilitou o processo de utilização e implementação da programação paralela utilizando GPU's com OpenCL e CUDA, assim o desenvolvedor poderá se concentrar mais no desenvolvimento da sua solução e não em como utilizar os recursos da GPU.

5.1 Contribuições

Além dos aspectos relacionados com o uso de uma arquitetura de baixo custo para processamento paralelo, as principais contribuições deste trabalho são:

- Facilidade de desenvolvimento: Com a utilização das bibliotecas o processo de desenvolvimento de soluções que utilizem as GPU's foi facilitado, porque o desenvolvedor não precisa mais verificar se existe alguma GPU, quantas e quais são essas placas existentes no computador utilizado. Assim o desenvolvedor irá se preocupar mais com a solução do seu problema do que com a construção do ambiente computacional.

- Transparência de inicialização da GPU: A inicialização dos contextos para cada GPU disponível tornou-se desnecessária, sendo que cada contexto armazena informações dos objetos em memória, o código-fonte e o código executável. A inicialização do contexto é necessária para que as informações sejam distribuídas na memória (desde os dados como as instruções) e a vantagem de tornar transparente essa configuração é a facilidade adquirida tanto para o programador, quanto para códigos já prontos que não precisam de manutenção em caso de troca (ou upgrade) da placa gráfica.

- Automatização na criação de filas: A criação das filas de execução de comandos ficou automatizada. Essas filas armazenam as funções ou os *kernels* que serão executadas pela GPU de acordo com o ordem de entrada nas filas.

- Transparência de compilação: Não é necessário determinar quando a compilação do código em OpenCL e CUDA irá ocorrer, possibilitando que o programador não se preocupe em quando será realizada a compilação, assim evitando o erro de compilar muito cedo ou muito tarde para enviar para o contexto da GPU.

- Transparência na criação do kernel da GPU: O desenvolvedor não precisa mais informar quando será criada o *kernel* ou a função a ser executada. O *kernel* para OpenCL ou a função para o CUDA, é qual método do código-fonte será executada. O programador ainda precisa determinar qual será executada, contudo não previsa determinar quando.

- Transparência no uso dos espaços de memória: A criação e a determinação do tamanho dos espaços em memória para as variáveis e dos seus respectivos tipos de dados foi automatizada, isso facilita muito a vida do programador pois antes era necessário calcular quanto de memória e quais os tipos dos dados que iriam utilizar na GPU, além de realizar a transferência de dados da máquina *host* para a GPU e vice-versa.
- Portabilidade entre placas: Não é necessário realizar os cálculos das fórmulas (8, 9, 10, 11, 12) para a execução em uma placa, pois em cada linguagem e GPU existem restrições com relação ao paralelismo suportado por cada um, permitindo assim a portabilidade entre placas sem a necessidade de reimplementar código para cada dispositivo.
- Diminuição de controles e erros sob responsabilidade do programador: A execução, o tratamento de erros, o retorno das respostas e a finalização da execução foram liberadas da responsabilidade do desenvolvedor, possibilitando assim uma menor quantidade de erros decorrentes da não liberação de memória na GPU.

5.2 Trabalhos Futuros

Para melhoria da solução gerada neste trabalho é necessário um maior aprofundamento nos conhecimentos de utilização das GPU's para complementar as bibliotecas, pois como foram desenvolvidas voltadas para cálculos com séries temporais alguns recursos das placas foram desprezados como por exemplo, funções com *strings* e imagens. Como um trabalho futuro pode-se adicionar as bibliotecas a utilização dos recursos das placas para imagens e *strings*, assim abrindo os horizontes de uso para a biblioteca. Pode-se também desenvolver um compilador de fórmulas para programação paralela afim de facilitar ainda mais os cálculos com dados ambientais.

A validação realizada com a dimensão fractal é importante, contudo mais cálculos relacionados com a Teoria da Complexidade devem ser implementados, o que direciona para a conversão da biblioteca TISEAN para a utilização da GPU, além de uma construção de uma interface gráfica para o mesmo, para facilitar sua utilização. O pacote TISEAN é um programa voltado para análise de séries temporais com métodos baseados na teoria de sistemas dinâmicos determinísticos não-lineares, ou teoria do caos (HEGGER *et al.*, 1999).

O armazenamento dos dados é um aspecto não tratado neste trabalho, contudo a utilização de um Sistema Gerenciador de Banco de Dados é imprescindível, assim é importante acoplar a biblioteca dos cálculos em um SGBD, como por exemplo PostgreSQL, afim de poder realizar os cálculos diretamente no banco de dados, através por exemplo do PLJava no PostgreSQL.

Para o programa da PGFA pode-se criar uma biblioteca que contenha a maioria dos cálculos utilizados afim de agilizar a obtenção dos resultados dos dados.

Mais uma possibilidade é a implementação do algoritmo *MapReduce* para a distribuição das tarefas utilizando as bibliotecas. *MapReduce* possui um modelo de programação simplista e tratamento automático de paralelização, agendamento e comunicação para processamento distribuído. Escondendo essas barreiras intrínsecas para a entrada os programadores começarem a desenvolver em grande escala para processamento de dados (BACKMAN *et al.*, 2012).

Referências Bibliográficas

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS 67 (Spring) Proceedings*, p. 483–485, 1967.

AUGUSTO, D. A.; BARBOSA, H. J. C. Accelerated parallel genetic programming tree evaluation with opencl. *Journal of Parallel and Distributed Computing*, 2012.

BACKMAN, N.; PATTABIRAMAN, K.; FONSECA, R.; CETINTEMEL, U. C-mr: continuously executing mapreduce workflows on multi-core processors. In: *Proceedings of third international workshop on MapReduce and its Applications Date*. New York, NY, USA: ACM, 2012. (MapReduce '12), p. 1–8. ISBN 978-1-4503-1343-8.

BAKER, G. L.; GOLLUB, J. P. Chaotic dynamics: an introduction. In: *Cambridge University Press*. New York, EUA: [s.n.], 1996. p. 256.

BALDWIN, C.; ABDULLA, G.; CRITCHLOW, T. Multi-resolution modeling of large scale scientific simulation data. In: *Proceedings of the twelfth international conference on Information and knowledge management*. New York, NY, USA: ACM, 2003. (CIKM '03), p. 40–48. ISBN 1-58113-723-0.

ECKMANN, J. P.; KAMPHORST, S. O.; RUELLE, D.; CILIBERTO, S. Recurrence plots of dynamical systems. *Europhysics Letter*, Europhysics Letter, v. 4, p. 973–977, 1987.

ELLIOTT, G. A.; ANDERSON, J. H. Globally scheduled real-time multiprocessor systems with gpus. *Real-Time Systems*, v. 48, n. 1, p. 34–74, 2012.

FIELDER-FERRARA, N.; PRADO, C. P. C. *Caos - Uma Introdução*. [S.l.]: EDGARD BLUCHER, 2011. ISBN 9788521200581.

FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 21, n. 9, p. 948–960, set. 1972. ISSN 0018-9340.

FOSTER, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201575949.

FURHT, B. Cloud computing fundamentals. In: FURHT, B.; ESCALANTE, A. (Ed.). *Handbook of Cloud Computing*. [S.l.]: Springer US, 2010. p. 3–19. ISBN 978-1-4419-6524-0.

GOMES, A. G.; VARRIALE, M. C. *Modelagem de ecossistemas: uma introdução*. [S.l.]: EditoraUFSM, 2001. ISBN 9788573910223.

GRASSBERGER, P.; PROCACCIA, I. Characterization of strange attractors. *Physical Review Letters*, v. 50, n. 5, p. 346–349, 1983.

GROUP, K. O. W. *The OpenCL specification, Version 1.1*. 2011. Disponível em http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

HEGGER, R.; KANTZ, H.; SCHREIBER, T. Practical implementation of nonlinear time series methods: The TISEAN package. *Chaos*, v. 9, p. 413, 1999.

HWANG, K. *Advanced computer architecture: parallelism, scalability, programmability*. [S.l.]: McGraw-Hill, 1993. (McGraw-Hill series in electrical and computer engineering: Computer engineering). ISBN 9780070316225.

HWANG, K.; BRIGGS, F. A. *Computer architecture and parallel processing*. [S.l.]: McGraw-Hill, 1984. (McGraw-Hill computer communications series). ISBN 9780070315563.

JCUDA. *Java bindings for CUDA*. 2012. Disponível em http://www.jcuda.org/.

JOCL. *Java bindings for OpenCL*. 2012. Disponível em http://www.jocl.org/.

LORENZ, E. N. Deterministic nonperiodic flow. *Journal of Atmospheric Sciences*, v. 20, p. 130–141, 1963.

MARWAN, N. A historical review of recurrence plots. *The European Physical Journal Special Topics*, Springer Berlin / Heidelberg, v. 164, n. 1, p. 3–12, nov. 2008. ISSN 1951-6401.

MARWAN, N.; CARMENROMANO, M.; THIEL, M.; KURTHS, J. Recurrence plots for the analysis of complex systems. *Physics Reports*, v. 438, n. 5-6, p. 237–329, jan. 2007. ISSN 03701573.

MELLO, G. J. *Dissertação de Mestrado*. 2010. Disponível em http://pgfa.ufmt.br/index.php?option=com_docman&task=doc_download&gid=166&Itemid=236.

MOORE, G. E. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, v. 86, n. 1, p. 82 –85, jan. 1998. ISSN 0018-9219.

NEVES, G. A. R. *Desenvolvimento de Estação Micrometeorológica com Armazenamento de Dados*. Tese (Doutorado) — Universidade de Mato Grosso, Cuiabá, MT, 2011.

NUSSENZVEIG, H. M. Complexidade e caos. *Revista Brasileira de Ensino de Física*, v. 22, n. 2, 2000.

NVIDIA. *NVIDIA OpenCL Jump Start Guide*. 2009a. Disponível em http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf.

NVIDIA. *OpenCL Programming for the CUDA Architecture*. 2009b. Disponível em http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingOverview.pdf.

NVIDIA. *NVIDIA CUDA C Programming Guide*. 2011. Disponível em http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

OWENS, J. D.; HOUSTON, M.; LUEBKE, D.; GREEN, S.; STONE, J. E.; PHILLIPS, J. C. GPU Computing. *Proceedings of the IEEE*, IEEE, v. 96, n. 5, p. 879–899, maio 2008. ISSN 0018-9219.

PAPAKONSTANTINOU, A.; GURURAJ, K.; STRATTON, J. A.; CHEN, D.; CONG, J.; HWU, W. M. W. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In: *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*. [S.l.: s.n.], 2009. p. 35 –42.

PEREIRA, M. F. L. *Um Modelo de Reconstrução Tomográfica 3D para Amostras Agrícolas com Filtragem de Wiener em Processamento Paralelo*. 148 p. Tese (Doutorado), 2007.

SARKAR, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989. ISBN 0262691302.

SEMCHEDINE, F.; MEDJKOUNE, L. B.; AıSSANI, D. Task assignment policies in distributed server systems: A survey. *Journal of Network and Computer Applications*, v. 24, p. 1123–1130, 2011.

STAROBA, J. *Parallel Performance Modeling, Prediction and Tuning*. 83 p. Tese (Doutorado), 2004.

TAKENS, F. Detecting strange attractors in turbulence. *Lecture Notes in Mathematics*, v. 898, p. 366–381, 1981.

THIEL, M. *Recurrences: exploiting naturally occurring analogues*. Tese (Doutorado) — University of Potsdam, Potsdam, 2004.

TORKESTANI, J. A. A new approach to the job scheduling problem in computational grids. *Cluster Computing*, Springer Netherlands, p. 1–10, 2011. ISSN 1386-7857. 10.1007/s10586-011-0192-5.

VALENTINI, G. L.; LASSONDE, W.; KHAN, S. U.; MIN-ALLAH, N.; MADANI, S. A.; LI, J.; ZHANG, L.; WANG, L.; GHANI, N.; KOLODZIEJ, J.; LI, H.; ZOMAYA, A. Y.; XU, C. Z.; BALAJI, P.; VISHNU, A.; PINEL, F.; PECERO, J. E.; KLIAZOVICH, D.; BOUVRY, P. An overview of energy efficiency techniques in cluster computing systems. *Cluster Computing*, p. 0 – 0, 2011.

WEIGEL, M. Performance potential for simulating spin models on gpu. *Journal of Computational Physics*, v. 231, n. 8, p. 3064–3082, 2012.

WOLF, A.; SWIFT, J.; SWINNEY, H.; VASTANO, J. Determining Lyapunov exponents from a time series. *Physica D: Nonlinear Phenomena*, v. 16, n. 3, p. 285–317, jul. 1985. ISSN 01672789.

YAMANAKA, A.; AOKI, T.; OGAWA, S.; TAKAKI, T. Gpu-accelerated phase-field simulation of dendritic solidification in a binary alloy. *Journal of Crystal Growth*, v. 318, n. 1, p. 40–45, 2011.

APÊNDICE A – Algoritmo em OpenCL

Algoritmo 11 Algoritmo na linguagem OpenCL

```
1:  __kernel void fractal(__global const double *dados,
2:  __global double *res,
3:  __global double *parametros,
4:  __global const double *dimensoes)
5:  {
6:     int gid = get_global_id(0);
7:      double soma = 0;
8:      int linha = parametros[1];
9:      int x = gid/linha;
10:     double rr = 0;
11:     rr = res[gid];
12:     double total = 0;
13:     double aux = 0;
14:     parametros[3] = parametros[3] + 1;
15:     for (int j = 0; j < (parametros[2] - 1 - (dimensoes[x] - 1) * parametros[0]); j++) {
16:         for (int i = j + 1; i < (parametros[2] - (dimensoes[x] - 1) * parametros[0]); i++) {
17:             soma = 0;
18:             aux += 1;
19:             for (int n = 0; n < dimensoes[x]; n++) {
20:                 int indice1 = j + n * parametros[0];
21:                 int indice2 = i + n * parametros[0];
22:                 double valor = dados[indice1] - dados[indice2];
23:                 soma = soma + valor * valor;
24:             }
25:             soma = sqrt(soma);
26:             if (rr > soma) {
27:              total += 2.0;
28:             }
29:         }
30:     }
31:     double tam = (parametros[2] - (dimensoes[x] - 1) * parametros[0]);
32:     res[gid] = total / (tam * (tam - 1));
33: }
```

APÊNDICE B – Algoritmo em CUDA

Algoritmo 12 Algoritmo na linguagem CUDA

```
1: extern "C"
2: __global__ void fractal(double *dados,double *res,double *parametros,double *dimen-
   soes) {
3:     int gid = blockIdx.x * blockDim.x + threadIdx.x;
4:     if (gid < parametros[3]){
5:         double soma = 0;
6:         int linhas = parametros[1];
7:         int x = gid/linhas;
8:         double rr = res[gid];
9:         res[gid] = 0;
10:        double aux = 0;
11:        double entro = 0;
12:        for (int j = 0; j < (parametros[2] - 1 - (dimensoes[x] - 1) * parametros[0]); j++) {
13:            for (int i = j + 1; i < (parametros[2] - (dimensoes[x] - 1) * parametros[0]); i++) {
14:                soma = 0;
15:                aux += 1;
16:                for (int n = 0; n < dimensoes[x]; n++) {
17:                    int indice1 = j + n * parametros[0];
18:                    int indice2 = i + n * parametros[0];
19:                    double valor = dados[indice1] - dados[indice2];
20:                    soma = soma + valor * valor;
21:                }
22:                soma = sqrt(soma);
23:                entro = 0;
24:                if (rr > soma)
25:                   entro = 2;
26:                res[gid] += entro;
27:            }
28:        }
29:        double tam = (parametros[2] - (dimensoes[x] - 1) * parametros[0]);
30:        res[gid] = res[gid]/(tam * (tam - 1));
31:    }
32: }
```