



UNIVERSIDADE FEDERAL DO RIO GRANDE  
PROGRAMA DE PÓS-GRADUAÇÃO EM  
MODELAGEM COMPUTACIONAL



# Computação de Alto Desempenho Aplicada a Modelagem Numérica de Fenômenos Atmosféricos

ANDERSON DUARTE SPOLAVORI

Dissertação apresentada ao Programa de Pós Graduação em Modelagem Computacional da Universidade Federal do Rio Grande (FURG) como requisito parcial à obtenção do título de Mestre em Modelagem Computacional.

Orientadora: Prof.<sup>a</sup> Dr.<sup>a</sup>Nisia Krusche  
Co-orientadora: Prof.<sup>a</sup> Dr.<sup>a</sup>Silvia Botelho

Rio Grande, janeiro de 2014.

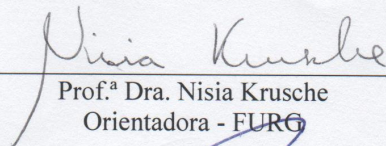
ANDERSON DUARTE SPOLAVORI

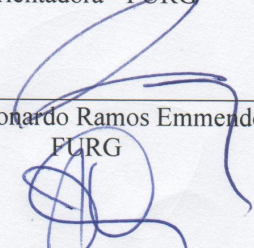
“COMPUTAÇÃO DE ALTO DESEMPENHO APLICADA À MODELAGEM NUMÉRICA DE FENÔMENOS ATMOSFÉRICOS”

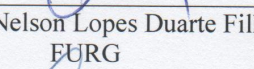
Dissertação apresentada ao Programa de Pós Graduação em Modelagem Computacional da Universidade Federal do Rio Grande -FURG, como requisito parcial para obtenção do Grau de Mestre. Área concentração: Modelagem Computacional.

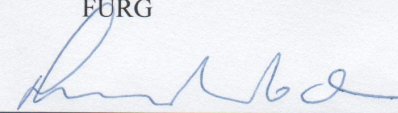
Aprovada em

BANCA EXAMINADORA

  
Prof.<sup>a</sup> Dra. Nisia Krusche  
Orientadora - FURG

  
Prof. Dr. Leonardo Ramos Emmendorfer  
FURG

  
Prof. Dr. Nelson Lopes Duarte Filho  
FURG

  
Prof.<sup>a</sup> Dra. Rosmeri Porfirio da Rocha  
USP

Rio Grande -RS  
2014

Dedico esta dissertação a minha  
esposa e a minha família.

## **AGRADECIMENTOS**

A minha orientadora, Nisia Krusche, pela amizade, atenção e dedicação.

A minha co-orientadora, Silvia Botelho, pela paciência abissal.

Aos meus pais, pelo incentivo e apoio.

A minha esposa pela compreensão e apoio incondicional.

Aos meus colegas do Núcleo de Física Ambiental que sempre me ajudaram.

A CAPES por financiar esta pesquisa.

E a todos que me ajudaram a alcançar mais um objetivo.

## RESUMO

A constante evolução da tecnologia disponibilizou, atualmente, ferramentas computacionais que eram apenas expectativas há 10 anos atrás. O aumento do potencial computacional aplicado a modelos numéricos que simulam a atmosfera permitiu ampliar o estudo de fenômenos atmosféricos, através do uso de ferramentas de computação de alto desempenho. O trabalho propôs o desenvolvimento de algoritmos com base em arquiteturas SIMT e aplicação de técnicas de paralelismo com uso da ferramenta OpenACC para processamento de dados de previsão numérica do modelo Weather Research and Forecast. Esta proposta tem forte conotação interdisciplinar, buscando a interação entre as áreas de modelagem atmosférica e computação científica. Foram testadas a influência da computação do cálculo de microfísica de nuvens na degradação temporal do modelo. Como a entrada de dados para execução na GPU não era suficientemente grande, o tempo necessário para transferir dados da CPU para a GPU foi maior do que a execução da computação na CPU. Outro fator determinante foi a adição de código CUDA dentro de um contexto MPI, causando assim condições de disputa de recursos entre os processadores, mais uma vez degradando o tempo de execução. A proposta do uso de diretivas para aplicar computação de alto desempenho em uma estrutura CUDA parece muito promissora, mas ainda precisa ser utilizada com muita cautela a fim de produzir bons resultados. A construção de um híbrido MPI + CUDA foi testada, mas os resultados não foram conclusivos.

## **ABSTRACT**

The ongoing evolution of technology yields, nowadays, computational tools that were mere expectations 10 years ago. The increase on computational potential applied to numerical models that simulate the atmosphere broadens the study of atmospheric phenomena due to high performance computing techniques. The present research intendeds to develop algorithms based on SIMT architectures and the use of paralel processing techniques with the use of the aplicative OpenACC to numerical data processing in the model Weather Research and Forecast, specifically the part the calculates cloud microphysics. This study has an intense interdisciplinary content, proposing the interaction between researchers of atmospheric modeling and of scientific computation. The influence of the computation, of the cloud microphysics calculation, on the temporal degradation of the model, was tested. Since the data input for the execution in the GPU was not large enough, the time needed to transfer the data from the CPU to the GPU was larger than the computation time in the CPU. Another determining factor was the addition of CUDA code in a MPI context, generating conditions of dispute over resources among the processors, once more degradating the time of execution. The suggestion of using directives to apply high performance computation in a CUDA structure seems very promising, but it still needs to be applied with caution to yield good results. The construction of a hybrid MPI + CUDA was also tested, but the results were uncertain.

## Sumário

CAPÍTULO 1: INTRODUÇÃO .....	14
<b>Objetivo</b> .....	16
CAPÍTULO 2: REVISÃO BIBLIOGRÁFICA .....	17
Paralelismo Computacional.....	17
Dependências .....	20
Tipos de Paralelismo .....	23
Linguagens e Diretivas para Programação Paralela .....	24
OpenMp .....	24
MPI.....	25
CUDA .....	26
CUDA™: uma arquitetura paralela de propósito geral .....	27
Hierarquia de Thread .....	28
Hierarquia de Memória.....	29
Programação Heterogênea .....	30
Capacidade Computacional .....	31
OpenACC.....	32
Paralelismo Automático.....	33
Computação de Alto Desempenho usando GPU .....	33
Modelagem Atmosférica.....	35
Microfísica de Nuvens .....	38
CAPÍTULO 3: METODOLOGIA.....	42
Análise de Erros .....	42
Análise de Complexidade de Algoritmos.....	42
Microfísica de Nuvem .....	44
Análise do Algoritmo da Microfísica de Nuvem .....	48
O ambiente para testes de performance .....	50
Aplicando técnicas de Paralelismo OpenACC .....	51
CAPÍTULO 4: RESULTADOS.....	54
Um Ensaio de Integração.....	63
BIBLIOGRAFIA .....	70

## LISTA DE SÍMBOLOS

$b$	perturbação da flutuabilidade
$C$	circulação
$e_s$	pressão de vapor de saturação
$f$	parâmetro de Coriolis
$\vec{k}$	versor do eixo vertical
$m$	fator do mapa
$N$	frequência de Brunt-Väisälä
$p_{hs}$	componente hidrostática da pressão na superfície
$p_{ht}$	componente hidrostática da pressão no topo da camada.
$p_0$	pressão de referência (tipicamente 1000 hPa)
$p_h$	componente hidrostática da pressão
$q$	umidade específica
$Q$	função de aquecimento
$R$	constante de Boltzmann
$T$	temperatura em Kelvin



## LISTA DE TABELAS

Tabela 1: Legenda da figura 9. ....	39
Tabela 2: Lista de estações de observações usadas no estudo comparativo entre diferentes microfísicas. ....	46
Tabela 3: Lista de directivas OpenACC. ....	53
Tabela 4: Demonstração dos resultados temporais, em segundos, da execução do modelo WRF, rodada completa, com e sem o uso da microfísica de nuvens WSM5. ....	58
Tabela 5: Demonstração dos resultados temporais, em segundos, da execução do modelo WRF, rodada de uma hora, com e sem o uso da microfísica de nuvens WSM5. ....	58
Tabela 6: Resultado, em milisegundos, das rodadas de teste de sensibilidade da variação do uso de diretivas e incremento do tamanho dos dados de entrada. Em verde o menor tempo de cada rodada e em vermelho o maior. ....	62
Tabela 7: Tomadas de tempo de processamento (ms), comparando CPU, GPU e GPU com uso de diretivas de controle de fluxo de dados, com múltiplas operações sob o mesmo dado. a) Matriz de dados de 200x200x27 elementos; b) Matriz de dados de 3200x3200x3 elementos.....	65

## LISTA DE FIGURAS

Figura 1: O tempo de execução e o aumento de velocidade de um programa com paralelismo. (Amdahl, 1952).....	20
Figura 2: Comparação entre a arquitetura da GPU e CPU, em verde, representa as Unidades Lógicas Aritiméticas, em laranja as memórias cachê e DRAM e em amarelo as unidades de controle. Fonte: NVIDIA CUDA C Programming Guide, 2012. ....	27
Figura 3: Grade de blocos de threads em verde, o grid de blocos, em amarelo os blocos de threads em laranja as threads. Fonte: NVIDIA CUDA C Programming Guide, 2012. ....	28
Figura 4: Hierarquia de memória. Em cinza, os tipos diferentes de memórias e seus níveis de atuação. Fonte: NVIDIA CUDA C Programming Guide, 2012. ....	29
Figura 5: Código serial executa no host, enquanto código paralelo executa no dispositivo. Fonte: NVIDIA CUDA C Programming Guide, 2012.....	30
Figura 6: A api é compatível posteriormente, mas não anteriormente. Fonte: NVIDIA CUDA C Programming Guide, 2012. ....	31
Figura 7: Descrição de operação entre CPU e GPU com o uso de OpenACC. Fonte: NVIDIA CUDA C Programming Guide, 2012. ....	33
Figura 8: Representação do aninhamento entre domínios: a) domínio pai com menor resolução; b) domínio filho (nest) com maior resolução. ....	37
Figura 9: Fluxograma dos processos do esquema WSM5. ....	39
Figura 10: Dominação assintótica de $f(n)$ sobre $g(n)$ . Fonte: CORMEN et al., 2009. ...	44
Figura 11; Fluxograma da execução dos testes de microfísica, ....	46
Figura 12: Demonstração da transcrição de código escrito para fluxograma de dados. ....	49
Figura 13: Trecho de código $O(n)$ para $O(1)$ .....	50
Figura 14: Precipitação total observada nas estações meteorológicas no período entre 16 de setembro de 2012 a 20 de setembro de 2012. ....	54
Figura 15: Dados observados e dados oriundos das saídas do modelo WRF com o uso de diferentes microfísicas de nuvem, a) Estação Meteorológica de Rio Grande. b) Estação Meteorológica de Porto Alegre. ....	55
Figura 16: Erro médio, dados observados versus dados modelo WRF com microfísicas 4, 8 e 10. Estação meteorológica de Rio Grande. ....	56

Figura 17: a) Chuva convectiva observada na saída do modelo com o uso da microfísica de nuvem WSM5. b) Chuva não convectiva observada na saída do modelo com o uso da microfísica de nuvem WSM5. c) Chuvas convectivas somadas as não convectivas observada na saída do modelo com o uso da microfísica de nuvem WSM5.....57

Figura 18: Tomada de tempo com adição nós, np significa o número de processadores a ser solicitado para a computação seguido do identificador da microfísica. a) Uso de GPU no processamento. b) Uso de GPU versus uso de CPU.66

## CAPÍTULO 1: INTRODUÇÃO

Dispõe-se, atualmente, com a constante evolução da tecnologia, de ferramentas computacionais que eram apenas expectativas há 10 anos atrás. O aumento do potencial computacional através de ferramentas de visualização e processamento abre um conjunto de possibilidades no que tange a interação homem máquina. Esta interação e o constante aperfeiçoamento dela é o caminho para o refinamento nos modelos computacionais e algoritmos, bem como para um melhor entendimento no estudo de fenômenos atmosféricos.

Inúmeras são as pesquisas realizadas em oceanografia, climatologia e meteorologia que envolvem modelagens numéricas e tratamento de grandes quantidades de dados. O estudo de problemas como previsão meteorológica, dispersão de poluentes e interação praia-oceano-atmosfera são exemplos típicos. Os modelos adotados para a resolução desses problemas, apesar de serem conhecidos há algum tempo, apenas recentemente ganharam uma dimensão aplicada dada a possibilidade de serem resolvidos com computação de alto desempenho.

O crescente número de dados a serem lidos, processados e interpretados infere em novas metodologias de análise, revelando a necessidade do uso de ferramentas de computação de alto desempenho.

É nesse contexto que o presente trabalho se propõe a avançar os limites do conhecimento focalizando-se no estudo, desenvolvimento e disponibilização de algoritmos e modelos computacionais capazes de extrair informações de grandes conjuntos de dados como contribuição às áreas acadêmicas didáticas e de pesquisa.

O estudo de técnicas de processamento de alto desempenho e a exploração de suas potencialidades, de forma mais específica, no uso direto em aplicações de modelos numéricos para o estudo de eventos atmosféricos, procurando contornar os problemas enfrentados por esse tipo de atividade.

Para que esses objetivos sejam alcançados, pretende-se fazer uso ostensivo de tecnologia e técnicas de paralelismo baseadas em Unidades de Processamento Gráfico (GPU). O desenvolvimento de algoritmos que utilizam esse tipo de tecnologia otimiza os processos computacionais, tornando-se necessário o desenvolvimento de aplicações baseadas em arquiteturas paralelas, mostrando ser um possível caminho para a melhoria do desempenho computacional.

Outra questão relevante é o fator custo-benefício. Historicamente, a utilização efetiva de computação de alto desempenho estava restrita a aplicações específicas devido à complexidade e custo dos sistemas de hardware e software envolvidos na sua implementação. Estas passavam pelo uso de máquinas sequenciais trabalhando em conjunto de forma a dividir e executar os trabalhos que podiam ser realizados em paralelo, os chamados clusters. Alavancado pela indústria de jogos, novas propostas de arquiteturas paralelas foram desenvolvidas, dentre elas, destacaram-se as GPUs NVIDIA. A proposta de investir em GPUs ao invés de várias CPUs – Unidade Central de Processamento barateia o custo de estações de processamento que fazem uso de programação de alto desempenho.

A maior facilidade de acesso à Computação de Alto Desempenho propiciada pelo uso de GPUs vem permitindo a aplicação de técnicas de paralelismo a novas categorias de problemas de ordem complexa, trazendo aportes em termos de tempo de processamento e qualidade das soluções obtidas. Destes pode-se citar a Modelagem Computacional de Fenômenos Físicos, Ambientais e Sociais, bem como a sua análise e visualização.

Dada a importância do tema tratado, este trabalho também é justificado pela aprovação de dois projetos de pesquisa encaminhados pelo grupo.

Em dezembro de 2007, o Programa de Pós-Graduação Modelagem Computacional da FURG propôs, em conjunto com o Programa de Pós-Graduação de Meteorologia da Universidade de São Paulo, o projeto, apresentado edital PROCAD/2007, denominado Modelagem Climática Regional Aplicada. Propunha-se realizar um conjunto de pesquisas combinando recursos humanos e infraestrutura de duas universidades brasileiras de regiões diferentes do Brasil, implementando diversas linhas de pesquisa de ponta relacionadas à modelagem climática regional aplicada, consolidando assim o Programa de Pós-Graduação em Modelagem Computacional da FURG. Este projeto proporciona a interação científica entre as instituições através do fluxo de docentes e discentes e, até o presente momento, já implementou modelo de previsão numérica de forma operacional, realizando inúmeras simulações e testes a fim de melhorar os resultados obtidos visando o desenvolvimento de formas de comunicação de previsão climática à população alvo.

Em julho de 2010, o Centro de Ciências Computacionais (C3) da FURG propôs o projeto apresentado ao edital MCT/CNPq Nº 09/2010 – PDI, Visualização Científica e de Engenharia aplicada a Modelagem Computacional de Sistemas Complexos

Naturais e Artificiais (KAIROS), que possuía como objetivos primários desenvolver algoritmos e técnicas que fariam uso efetivo de arquiteturas gráficas, interface que permitisse ao usuário interagir com ambiente de visualização de forma imersiva e interativa e fortalecer o aspecto interdisciplinar, estreitando o vínculo entre pesquisadores de diferentes áreas associadas aos problemas a serem modelados.

Assim, parte da motivação para esta dissertação parte dos desafios apontados nestes dois projetos constituindo-se na busca por soluções que implementem processos, dentro do escopo do modelo numérico, que façam uso de computação paralela de GPUs.

## **Objetivo**

O objetivo deste projeto é averiguar o uso de tecnologias avançadas de processamento para realização de computação científica na análise de fenômenos atmosféricos. Como produto final desse trabalho, espera-se desenvolver algoritmos com base em arquiteturas SIMT (Single Instruction Multiple Threads) para processamento de dados de previsão numérica. O termo SIMT foi criado pela NVIDIA para descrever suas arquiteturas em GPUs, contudo, são fundamentalmente SIMD (Single Instruction Multiple Data). Esta proposta tem forte conotação interdisciplinar e pretende estreitar as relações entre pesquisadores das diferentes áreas do conhecimento.

Como objetivos específicos, espera-se:

- verificar as características e potenciais da aplicação de arquiteturas SIMT;
- selecionar um modelo de previsão atmosférica adequado para a região;
- desenvolver e avaliar pelo menos um algoritmo com base em arquiteturas SIMT para o modelo selecionado fazendo uso da API (*Programming Application Interface*) OpenACC;

## CAPÍTULO 2: REVISÃO BIBLIOGRÁFICA

### Paralelismo Computacional

Tradicionalmente, a grande maioria das aplicações de software é escrita como programas sequenciais, conforme descrito por Von Neumann (1945), com algoritmos implementados como um fluxo serial de instruções de aritmética básica, lógica e a entrada e saída de dados, resolvidas uma a uma por uma Unidade Central de Processamento (CPU) de um computador.

O impulso de aumentar a velocidade de execução do software baseando-se apenas nos avanços alcançados no hardware tem diminuído desde 2003, principalmente devido à questões de consumo de energia e dissipação de calor. Nesta avaliação, o conceito de *clock* torna-se relevante. *Clock* é um sinal usado para coordenar as ações de dois ou mais circuitos eletrônicos, em eletrônica e especialmente em circuitos digitais síncronos. As questões de consumo e dissipação limitam o aumento da frequência de *clock* (ciclos por segundo) e o nível de tarefas que podem ser realizadas em cada período de *clock* dentro de uma única CPU.

Em termos gerais, o tempo de execução de um programa corresponde ao número de instruções multiplicado pelo tempo médio de execução por instrução. Ao aumentar a frequência de processamento de um computador, reduz-se o tempo médio para executar uma instrução, diminuindo assim o tempo para execução dos programas.

Entretanto, o consumo de energia de um chip é dado por (1)

$$P = C.V^2.F \quad (1)$$

onde P é a energia, C é a capacitância sendo trocada por ciclo de *clock* (proporcional ao número de transistores cujas entradas mudam), V é a tensão e F o clock, representando a frequência do processador em ciclos por segundo. Aumentar a frequência é aumentar a energia usada em um processador. Em 2004, esse aumento de consumo levou a Intel a cancelar seus modelos de processadores Tejas e Jayhawk,

geralmente citado como o fim da frequência de processamento como paradigma predominante nas arquiteturas de computador.

Microprocessadores baseados em uma única CPU, como os da família Intel Pentium ou da família AMD Opteron, impulsionaram aumentos de desempenho e reduções de custo nas aplicações de computadores por mais de duas décadas. Esses equipamentos trouxeram bilhões de operações de ponto flutuante por segundo (GFLOPS) para computadores convencionais e centenas de GFLOPS para servidores em cluster. A constante evolução permite que o software de aplicação ofereça mais funcionalidades, uma interface com o usuário mais rica em elementos e gere melhores resultados.

A maioria dos fabricantes de microprocessadores passaram para modelos em que várias unidades de processamento, conhecidas como núcleos ou cores, são usadas em cada chip para aumentar o poder de processamento. Essa mudança exerceu grande impacto sobre a comunidade de desenvolvimento de software (Sutter 2005).

Um dos conceitos fundamentais na computação é o de fila ou encadeamento de execução, em inglês *thread*, que é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente. Uma thread permite, por exemplo, que o usuário de um programa utilize uma funcionalidade do ambiente enquanto outras linhas de execução realizam outros cálculos e operações.

Em hardwares equipados com uma única CPU, cada thread é processada de forma aparentemente simultânea, pois a mudança entre uma thread e outra é feita de forma tão rápida que para o usuário isso está acontecendo paralelamente.

Em hardwares com múltiplos CPUs ou multi-cores, as threads são processadas realmente de forma simultânea. Os sistemas que suportam apenas uma única thread são chamados de *monothread*, enquanto que os sistemas que suportam múltiplas threads são chamados de *multithread*.

A expectativa de um determinado programa sequencial executar de forma mais rápida com o incremento da capacidade de processamento não é válida, uma vez que esse programa roda em apenas um núcleo que não se tornará muito mais rápido do que os disponíveis atualmente. Sem a melhoria no desempenho, os desenvolvedores de



aplicação não poderão adicionar novas funcionalidades e recursos em seus programas à medida que novos microprocessadores sejam introduzidos. Partindo desse princípio, espera-se que os novos softwares gozem das melhorias de desenvolvimento baseando sua arquitetura em códigos que fazem uso ostensivo de paralelismo, onde vários threads de execução cooperam para completar o trabalho mais rapidamente. Esse novo incentivo escalado para o desenvolvimento de programa paralelo tem sido conhecido como Revolução da Concorrência (Sutter, 2005).

A prática da programação paralela não é nova, ela tem sido usada há décadas. Contudo, essa prática só funcionava em supercomputadores muito caros, nos quais, somente algumas aplicações justificavam o seu uso. Atualmente, todos os novos microcomputadores são computadores paralelos e a quantidade de aplicações que necessitam fazer uso dessa tecnologia também aumentou, logo, torna-se necessário que os desenvolvedores aprendam a programar de forma paralela.

Teoricamente, o aumento da velocidade com o paralelismo deveria ser linear, de maneira que dobrando a quantidade de elementos de processamento, reduz-se pela metade o tempo de execução. No entanto, poucos algoritmos paralelos atingem essa situação considerada ideal. A maioria dos algoritmos possui um aumento de velocidade quase linear para poucos elementos de processamento, tendendo a um valor constante para uma grande quantidade de elementos. O aumento de velocidade de um algoritmo em uma plataforma paralela é dado pela lei de Amdahl (Gene Amdahl, 1952). Ele afirma que apenas uma pequena porção do programa que não pode ser paralelizada, limitará o aumento de velocidade geral disponível com o paralelismo.

A figura 1 mostra o gráfico do tempo de execução versus aumento de velocidade de um programa com paralelismo. A linha azul demonstra o caso ideal, enquanto a curva lilás indica o aumento real. De maneira semelhante, a curva amarela indica o tempo de execução no caso ideal enquanto a vermelha o tempo real. Para qualquer problema que exija uma grande quantidade de computação, sempre haverá partes sequenciais e paralelizáveis. A relação entre elas é definida por (2)

$$S = \frac{1}{1-P} \quad (2)$$

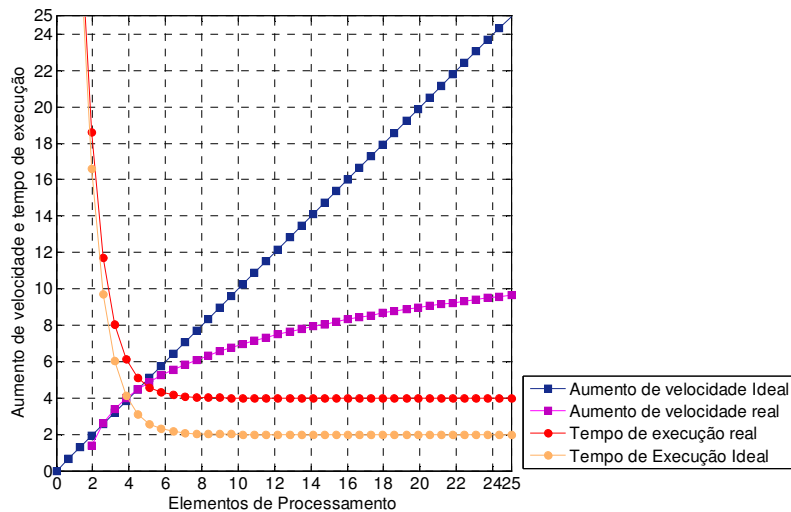


Figura 1: O tempo de execução e o aumento de velocidade de um programa com paralelismo. (Amdahl, 1952)

onde  $S$  é o aumento de velocidade do programa e  $P$  a parte paralelizável. No caso da porção paralelizável do programa ser de 20% do tempo de execução, não se pode obter mais que 20 vezes de aumento de velocidade, independente de quantos processadores são adicionados.

A lei de Gustafson está relacionada com a de Amdahl, descrita conforme (3)

$$S(P) = P - \alpha(P - 1) \tag{3}$$

onde  $P$  é o número de processadores,  $S$  é o aumento de velocidade e  $\alpha$  é a parte não paralelizável do processo. A lei de Amdahl assume um tamanho de problema fixo e que o tamanho da seção sequencial é independente do número de processadores, enquanto a lei de Gustafson não parte desse princípio.

## Dependências

Para a implementação de algoritmos paralelos, é fundamental o entendimento da dependência de dados. Nenhum programa pode executar mais rápido que a execução do seu caminho crítico, ou seja, a maior cadeia de cálculos dependentes. Quando um cálculo depende do cálculo anterior da cadeia, ele precisa ser executado

de forma sequencial. Contudo, a maioria dos algoritmos não é composta de longas cadeias de cálculos dependentes, geralmente existem muitas oportunidades de executar cálculos independentes em paralelo.

Assumindo-se os fragmentos de programa  $P_i$  e  $P_j$ . As condições de Bernstein (1966) descrevem quando os dois fragmentos são independentes e podem ser executados de forma paralela.

Para  $P_i$ , assume-se que  $I_i$  são todas as variáveis de entrada e  $O_i$  todas as variáveis de saída, e o mesmo para  $P_j$ .  $P_i$  e  $P_j$  são independentes se:

$$I_j \cap O_i = \emptyset$$

$$I_i \cap O_j = \emptyset$$

$$O_i \cap O_j = \emptyset$$

A primeira condição diz respeito à dependência de fluxo, correspondendo ao primeiro fragmento podendo produzir ou não um resultado usado pelo segundo. A segunda condição é a antidependência, quando o primeiro fragmento sobrescreve uma variável utilizada na segunda expressão. A terceira representa a dependência de saída. Quando duas variáveis escrevem no mesmo local, a saída final deve vir do segundo fragmento.

### **Taxonomia de Flynn**

A taxonomia de Flynn é uma classificação de arquiteturas de computadores proposta por Michael J. Flynn em 1966. De acordo com o autor, foram definidas quatro classificações baseadas no número de instruções concorrentes e no fluxo de dados disponíveis na arquitetura. São elas:

**Single Instruction, Single Data stream (SISD):** Um computador sequencial que não explora nenhum tipo de paralelismo em seu conjunto de instruções ou fluxo de dados. Dotado de uma única unidade de controle (UC) e instrução de fluxo para memória, executando instruções uma a uma em sua unidade lógica aritmética (ULA).

**Single Instruction Multiple Data (SIMD):** Trata-se de uma arquitetura que explora a execução de uma instrução simples sobre múltiplos conjuntos de dados

simultaneamente. O primeiro registro do uso de instruções SIMD data do início da década de 60 na Westing House Electric Corporation durante o projeto Solomon. O objetivo era aumentar o desempenho baseado na quantidade de co-processadores matemáticos sob o controle de uma única CPU. A CPU enviava uma instrução simples, comum a todas as ULAs, para executar um conjunto de dados, agrupados em um vetor, em diferentes pontos em um único ciclo de *clock*. Em 1962 o projeto foi cancelado e outros projetos que se seguiram não obtiveram sucesso por trabalhar com vetores muito pequenos. A técnica só foi usada com eficiência em 1976, por Seymour Cray, fundador da Cray Research. O super computador Cray-1 foi o primeiro a se beneficiar de uma arquitetura vetorial. Ele possuía registradores vetoriais, que proporcionavam um processamento muitas vezes mais rápido que seus concorrentes vetoriais. Após seis meses de testes, foi adquirido pelo National Center for Atmospheric Research (NCAR). O modelo SIMD é muito parecido com o processamento vetorial, diferindo somente pelo fato de que as unidades funcionais em máquinas vetoriais são projetadas para funcionar na forma de pipeline, enquanto em máquinas SIMD os elementos são operados todos de uma só vez (Patterson, 2005). SIMD é comumente aplicável em tarefas como ajuste de contraste em imagens digitais ou ajustes de volume em áudios digitais. As CPUs mais modernas já possuem instruções do tipo SIMD para melhorar o desempenho no uso de multimídia.

**Multiple Instruction Single Data (MISD):** Essa arquitetura consiste de uma unidade de controle com N unidades de processamento. Todo o processamento está sobre o controle de um único fluxo de instruções, mas opera sobre N fluxos de dados, Esta arquitetura pode ser utilizada para o caso de diferentes algoritmos atuarem em um mesmo conjunto de dados, entretanto, existem poucos registros dessa arquitetura.

**Multiple Instruction Multiple Data (MIMD):** Máquinas utilizando MIMD têm um número de processadores que funcionam de maneira assíncrona e de forma independente. A qualquer momento, diferentes processadores podem estar executando instruções diferentes em partes diferentes dos dados. Arquiteturas MIMD podem ser usadas nas

áreas de simulação, modelagem e como interruptores de comunicação. Máquinas MIMD trabalham com memória compartilhada ou distribuída.

## **Tipos de Paralelismo**

Existem vários tipos de paralelismo em diferentes níveis hierárquicos. A relevância de caracterizá-los provém da necessidade de especificar quais os tipos foram empregados durante o desenvolvimento desta pesquisa.

**Na Instrução:** Em sua essência, um programa de computador é um fluxo de instruções executadas por um processador. Tais instruções podem ser reordenadas e combinadas em grupos que então são executados em paralelo sem mudar o resultado do programa. Isso é conhecido por paralelismo em instrução (Culler, 1999). Avanços nessa técnica dominaram a arquitetura de computador de meados da década de 1980 até meados da década de 1990. Processadores modernos possuem pipeline com múltiplos estágios. Cada estágio corresponde a uma ação diferente que o processador executa em determinada instrução; um processador com um pipeline de N estágios pode ter até N diferentes instruções em diferentes estágios de execução. O exemplo canônico é o processador RISC, com cinco estágios: instruction fetch, decode, execute, memory access e write back. O processador Pentium 4 possui um pipeline de 35 estágios ( Patt, 2004). Alguns processadores podem lidar com mais de uma instrução por vez, além do paralelismo em instrução obtido com o pipeline. São conhecidos como processadores superescalares. As instruções podem ser agrupadas somente se não há dependência de dados entre elas. O algoritmo de marcador e o algoritmo de Tomasulo são duas das técnicas mais usadas para implementar reordenação de execução e paralelismo em instrução.

**No Dado:** O paralelismo em dado é inerente a laços de repetição, focando em distribuir o dado por diferentes nós computacionais para serem processados em paralelo. Diversas aplicações científicas e de engenharia apresentam esse tipo de paralelismo. Citando como exemplo os modelos numéricos WRF e RegCM, que distribuem parcelas

de seus dados de entrada para diferentes nós para serem processados de forma independente. Uma dependência por laço é a dependência de uma iteração do laço com a saída de uma ou mais iterações anteriores, uma situação que impossibilita a paralelização de laços.

**Na Tarefa:** Paralelismo em tarefa é a característica de um programa paralelo em que diferentes cálculos são realizados no mesmo ou em diferentes conjuntos de dados. Isso contrasta com o paralelismo em dado, em que o mesmo cálculo é feito em diferentes conjuntos de dados. Paralelismo em tarefa geralmente não é escalonável com o tamanho do problema.

## Linguagens e Diretivas para Programação Paralela

Geralmente, as linguagens dividem-se em classes de plataformas, baseadas nas premissas sobre a arquitetura de memória usada, incluindo memória compartilhada, memória distribuída ou memória compartilhada e distribuída. Serão descritas sucintamente a OpenMP (*The OpenMP® API specification for parallel programming*, 2013, doravante denominado OMP), a MPI (*Open Source High Performance Computing* 2013, doravante denominado MPIbb), a *Compute Unified Device Architecture*, conhecida por CUDA (NVIDIA *CUDA Home Page*, 2013, doravante denominado NVIDIAH) e a OpenACC (*OpenACC Directives for accelerators*, 2013, doravante denominado OPACC).

### OpenMp

OpenMP é uma implementação de *multithreading*, um método de paralelização no qual o "*master thread*" (uma série de instruções executadas consecutivamente) *forks* ("bifurca") um específico número de *threads* escravos e uma tarefa é dividida entre eles. Os *threads* são então executados simultaneamente, com o ambiente de execução distribuindo os *threads* para diferentes processadores.

A parte de código que é criada para funcionar em paralelo é marcada de acordo com uma diretriz pré-processador que criará os *threads* para formar a secção antes de

ser executada. Cada *thread* tem um "id" (endereço) anexado a ele. O *thread* "id" é um número inteiro e o *master thread* possui o id "0". Após a execução do código em paralelo, os *threads* retornam ao *master thread*, o qual continua progressivo até o fim do programa.

Por padrão, cada *thread* executa a seção do código paralelo independentemente. "Construções de Partilha" pode ser usada para dividir uma tarefa entre os *threads* de modo que cada *thread* execute a sua parte do código atribuído. Ambos, o paralelismo de tarefas e o paralelismo de dados, podem ser conseguidos utilizando OpenMP desta forma.

Em tempo de execução são distribuídas as *threads* aos processadores de acordo com a utilização, o desempenho da máquina e outros fatores. O número de *threads* pode ser fixado pelo ambiente de execução baseado em variáveis de ambiente ou um código usando funções.

## **MPI**

*Message Passing Interface* (MPI) é um padrão para comunicação de dados em computação paralela. Existem várias modalidades de computação paralela, e dependendo do problema que se está tentando resolver, pode ser necessário passar informações entre os vários processadores ou nodos de um cluster, e o MPI oferece uma infraestrutura para essa tarefa.

No padrão MPI, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Inicialmente, na maioria das implementações, um conjunto fixo de processos é criado. Porém, esses processos podem executar diferentes programas. Por isso, o padrão MPI é algumas vezes referido como MPMD (*multiple program multiple data*). Elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga. Dado o fato do número de processos no MPI ser normalmente fixo, neste texto é focado o mecanismo usado para comunicação de dados entre processos. Os processos podem usar mecanismos de comunicação ponto a ponto (operações para enviar mensagens

de um determinado processo a outro). Um grupo de processos pode invocar operações coletivas (*collective*) de comunicação para executar operações globais. O MPI é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores (*communicator*) que permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna.

O objetivo de MPI é prover um amplo padrão para escrever programas com passagem de mensagens de forma prática, portátil, eficiente e flexível.

## **CUDA**

CUDA é uma plataforma de computação paralela e um modelo de programação desenvolvido pela NVIDIA (NVIDIAH). Ela permite aumentos significativos de performance computacional ao aproveitar a potência da unidade de processamento gráfico (GPU).

Com milhões de GPUs habilitadas para CUDA já vendidas até hoje, os desenvolvedores de software e pesquisadores estão descobrindo usos amplamente variados para a computação com GPU CUDA. As áreas são muitas, como por exemplo, na identificação de placas ocultas em artérias, onde GPUs são utilizadas com o objetivo de simular o fluxo sanguíneo e identificar placas arteriais ocultas sem fazer uso de técnicas de imageamento invasivas ou cirurgias exploratórias.

Impulsionado pela alta demanda do mercado por computação em tempo real e gráficos 3D de alta definição, a unidade de processamento gráfico evoluiu para um processador com alta escalabilidade de núcleos, elevado poder computacional e largura de banda de memória.

A razão da discrepância na capacidade de operações de ponto flutuante entre a CPU e a GPU deve-se ao fato de que a última é especializada em computação intensiva, altamente paralela (exatamente o que a renderização de gráficos necessita) e portanto, projetada de tal forma que mais transistores são dedicados ao processamento de dados ao invés de cache de dados e controle de fluxo, como ilustrado na figura 2.



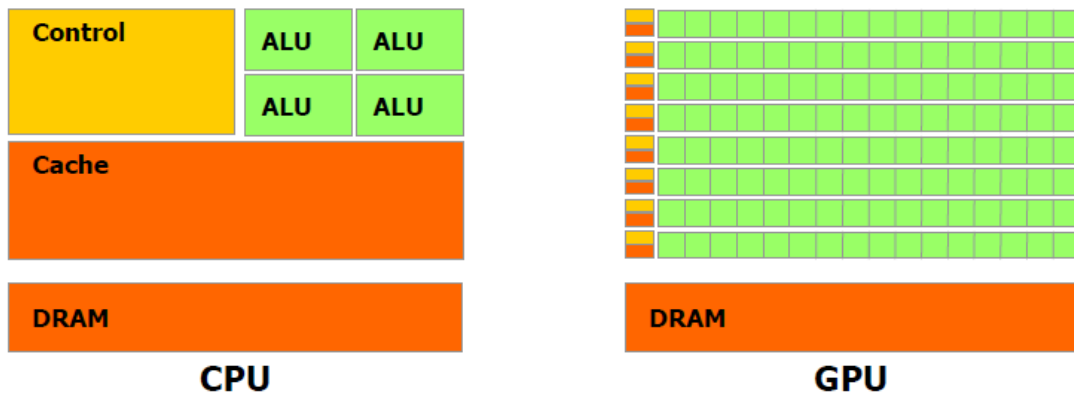


Figura 2: Comparação entre a arquitetura da GPU e CPU, em verde, representa as Unidades Lógicas Aritiméticas, em laranja as memórias cachê e DRAM e em amarelo as unidades de controle. Fonte: NVIDIA CUDA C Programming Guide, 2012.

Especificamente, a GPU é especialmente projetada para resolver problemas que podem ser expressos como computações de dados em paralelo - o mesmo programa executado em diversos elementos de dados em paralelo com alta intensidade. Como a mesma operação é executada para cada elemento de um conjunto de dados, existe menor necessidade de um controle de fluxo sofisticado e porque é executada em vários elementos de dados e tem alta intensidade aritmética, a latência de acesso à memória pode esconder-se com os cálculos ao invés de grandes memórias cachê.

### **CUDA™: uma arquitetura paralela de propósito geral**

Em novembro de 2006, a NVIDIA CUDA™ introduziu uma arquitetura de computação paralela de propósito geral com um novo modelo de programação e arquitetura de conjunto de instruções que utiliza o motor de computação paralela em GPUs NVIDIA para resolver problemas computacionais complexos de uma maneira mais eficiente do que a GPU.

CUDA vem com um ambiente de software que permite aos desenvolvedores usar C como uma linguagem de programação de alto nível. Outras linguagens e diretivas também são suportadas, tais como FORTRAN, DirectCompute, OpenCL, e OpenACC. CUDA C estende C, permitindo ao programador definir funções em C, chamadas de núcleos, que quando chamadas, são executadas N vezes em paralelo

por N diferentes *threads* CUDA, diferente de funções regulares C que são executadas uma só vez.

## Hierarquia de Thread

Por conveniência, *ThreadIdx* é um vetor de três componentes, de modo que as *threads* podem ser identificadas em um, dois ou três índices de *thread* formando um uni, bi ou tri-dimensional bloco de *threads*. Isso provê uma forma natural de invocar computação através dos elementos de um domínio como um vetor, matriz ou volume.

Existe um limite para o número de threads por bloco, já que todas as *threads* de um bloco devem residir no mesmo núcleo de processador e compartilhar os recursos limitados de memória desse núcleo. O número total de *threads* é igual ao de *threads* por bloco vezes o número de blocos.

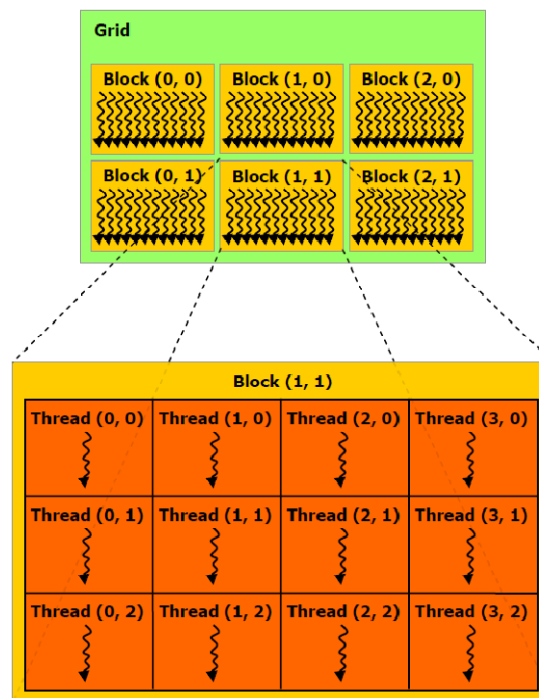


Figura 3: Grade de blocos de threads em verde, o grid de blocos, em amarelo os blocos de threads em laranja as threads. Fonte: NVIDIA CUDA C Programming Guide, 2012.

Os blocos são organizados em uma rede de blocos, como mostra figura 3. O número de blocos de threads em uma grade normalmente é ditada pelo tamanho dos

dados que estão sendo processados ou o número de processadores do sistema, que podem ser excedidos em muito.

## Hierarquia de Memória

*Threads* CUDA podem acessar dados de vários espaços de memória durante sua execução, conforme ilustrado pela figura 4. Cada um deles possui uma memória local privada. Cada bloco de *threads* possui memória visível a todas as *threads* do mesmo bloco com o mesmo tempo de vida. Todas as *threads* tem acesso a mesma memória global.

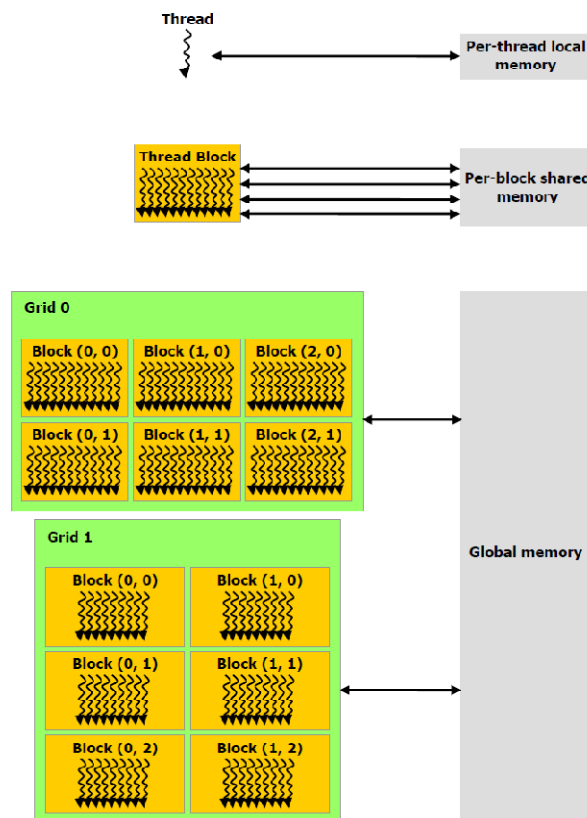


Figura 4: Hierarquia de memória. Em cinza, os tipos diferentes de memórias e seus níveis de atuação. Fonte: NVIDIA CUDA C Programming Guide, 2012.

Há também dois espaços de memória, somente para leitura, acessíveis por todas as *threads*: O espaço de constantes e o de textura. As memórias globais, de constantes e de textura são otimizadas para diferentes tipos de uso de memórias.

## Programação Heterogênea

O modelo de programação CUDA assume que as *threads* CUDA executam em

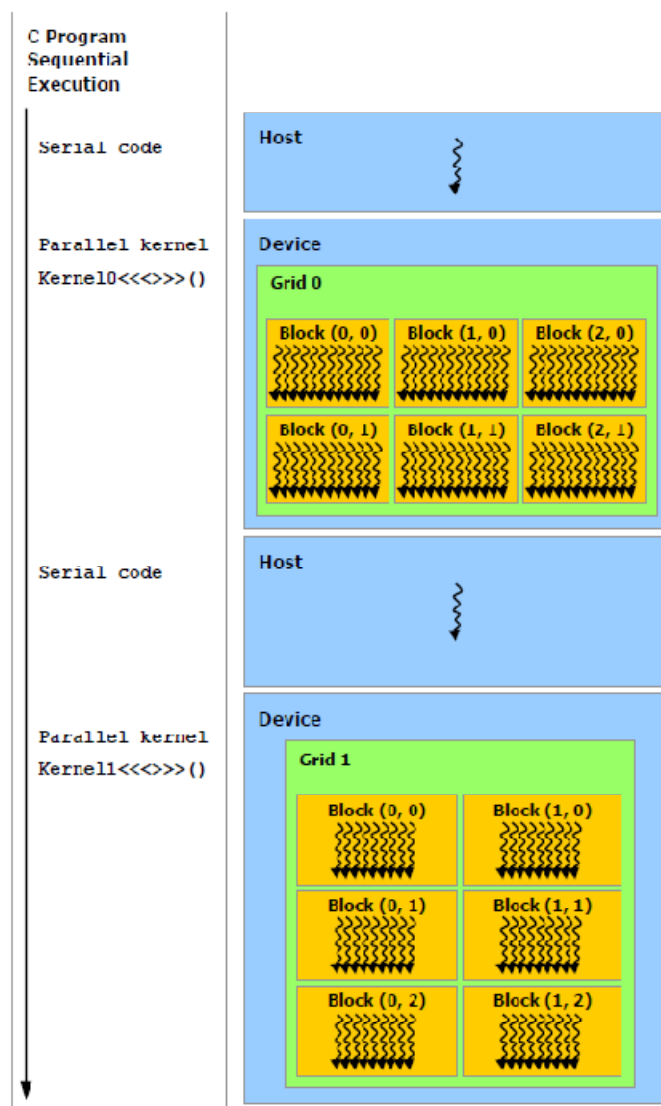


Figura 5: Código serial executa no host, enquanto código paralelo executa no dispositivo. Fonte: NVIDIA CUDA C Programming Guide, 2012.

um dispositivo separado (GPU) que opera como um co-processador para o *host* (anfitrião ou máquina a qual a GPU esta ligada). Na figura 5 demonstra um exemplo onde o núcleo executa na GPU e o resto do programa executa em uma CPU (*host*). Ambos possuem seus próprios e independentes espaços de memória DRAM (Dynamic random Access memory ou memória dinâmica de acesso randômico), referenciadas como *host memory* (memória do anfitrião) e *device memory* (memória do dispositivo). Portanto, o programa gerencia os espaços de memória global, de constantes e de textura, visíveis para o núcleo através de chamadas para o CUDA em tempo de execução. Isso inclui a alocação e desalocação de memória do dispositivo, bem como a transferência de dados entre o *host* e o *device memory*.

### Capacidade Computacional

A capacidade computacional do dispositivo é definida pelo maior e pelo menor número de revisão. Dispositivos com o mesmo número de revisão alto possuem a mesma arquitetura. O maior é 3 para dispositivos baseados na arquitetura *Kepler*, 2 para *Fermi* e 1 para *Tesla*. O menor número corresponde a incrementos e melhorias no núcleo da arquitetura, possivelmente incluindo novas características. Por exemplo, 3.2 e 3.1, pertencem a mesma arquitetura, o que não ocorre com 1.2.

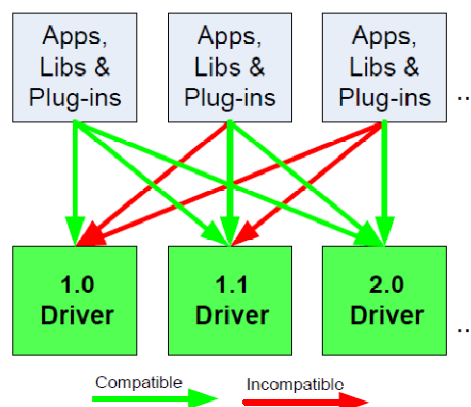


Figura 6: A api é compatível posteriormente, mas não anteriormente. Fonte: NVIDIA CUDA C Programming Guide, 2012.

Essa discussão torna-se relevante, uma vez que para possíveis soluções de paralelismo, a GPU utilizada nessa pesquisa, não suporta algumas funcionalidades exigidas pelas melhores soluções disponíveis atualmente. A figura 6 demonstra como o sistema de compatibilidade funciona. Bibliotecas são compatíveis em versões posteriores, mas nunca em anteriores.

## OpenACC

O OpenACC descreve um conjunto de diretivas de compilador para especificar loops e regiões de código padrão C, C++ e Fortran para ser descarregado, a partir de uma CPU *host*, para uma GPU, proporcionando portabilidade entre sistemas operacionais, processadores e aceleradores.

As diretrizes e o modelo de programação definidos pelo OpenACC permitem aos programadores criar programas em alto nível sem a necessidade de inicializar explicitamente a GPU e gerenciar dados ou transferências de programas entre o *host* e o acelerador.

Todos estes detalhes estão implícitos no modelo de programação e são geridos pelos compiladores em tempo de execução pela API OpenACC. O modelo de programação permite ao programador aumentar a informação disponível aos compiladores, incluindo especificação da localização dos dados do acelerador, orientação sobre mapeamento de loops, e detalhes relacionados ao desempenho.

A execução da implementação alvo da API do OpenACC é *host-dirigido*, transferindo a execução para o dispositivo acelerador anexado, uma GPU por exemplo. A maior parte da aplicação executa no *host*. Regiões que necessitam de computação intensiva são passadas para o dispositivo acelerador sob o controle do *host*. O dispositivo executa regiões paralelas que normalmente contém *loops* de trabalhos compartilhados ou regiões kernel (núcleo).

Mesmo em regiões alvo do acelerador, o *host* pode orquestrar a execução de alocação de memória no dispositivo, iniciando a transferência de dados, enviando o código para o acelerador, passando argumentos para a região da computação,

ordenando a fila do código do dispositivo, esperar pela conclusão, transferir os resultados de volta para o host e deslocar a memória. A figura 7 retirada do site da NVIDIA exemplifica a interação CPU-GPU e a definição da parcela de código que será executada pela GPU.

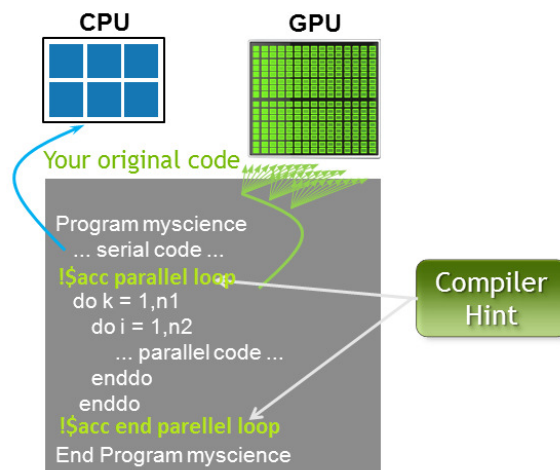


Figura 7: Descrição de operação entre CPU e GPU com o uso de OpenACC. Fonte: NVIDIA CUDA C Programming Guide, 2012.

### Paralelismo Automático

A habilidade de um compilador realizar paralelismo automático de um programa sequencial é um dos assuntos mais estudados no campo de computação paralela. Entretanto, apesar de décadas de trabalho por pesquisadores de compiladores, pouco sucesso se teve na área (Shen, 2005). Linguagens de programação paralelas de amplo uso continuam explícitas, isto é, requerendo a programação paralela propriamente dita ou no máximo parcialmente implícitas, em que o programador fornece ao compilador diretivas para o paralelismo.

### Computação de Alto Desempenho usando GPU

Desde 2003, a indústria de semicondutores tem estabelecido duas trajetórias principais para o projeto de microprocessador (Hwu, 2008). A trajetória de múltiplos núcleos, que é largamente utilizada e busca maximizar a velocidade de execução de

programas sequenciais, e a trajetória baseada em muitos núcleos que tem como objetivo a execução de várias aplicações paralelas. Um exemplo desse tipo de arquitetura é a GPU GeForce GTX da NVidia, com 240 núcleos, cada um deles sendo maciçamente *multithreaded*. Processadores com muitos núcleos, especialmente as GPUs, tem liderado a corrida do desempenho em ponto flutuante e é buscando esse ganho, que iniciou-se um estudo completo de como fazer uso dessa arquitetura.

Embora o ritmo de melhoria de desempenho de microprocessadores de uso geral tenha desacelerado, o das GPUs continua a melhorar incessantemente.

Existe uma grande diferença de desempenho entre a execução paralela e a sequencial, essa diferença já motivou muitos desenvolvedores de aplicação a mudarem seus códigos procurando fazer uso de GPUs. Alguns dos motivos dessa grande diferença de desempenho devem-se as diferenças entre as filosofias de projeto fundamentais nos dois tipos de processadores. A CPU é otimizada para desempenho de código sequencial. Ela utiliza lógica de controle sofisticada para permitir que instruções de uma única *thread* de execução sejam executadas em paralelo ou mesmo fora de sua ordem sequencial, enquanto mantém a aparência de execução sequencial. Mais importante, grandes memórias cachê são fornecidas para reduzir as latências de acesso a instruções e dados de aplicações. Nem a lógica de controle nem as memórias cachê contribuem para a velocidade máxima de cálculo.

A largura de banda da memória é outra questão importante. Os chips gráficos tem operado com cerca de 10 vezes a largura de banda dos chips de CPU disponíveis. Por exemplo o chip GT200 da NVIDIA, admite cerca de 150GB/s, a largura de banda de memória do sistema microprocessador provavelmente não ultrapassará os 50GB/s nos próximos anos.

Deve ficar claro que as GPUs são projetadas como mecanismos de calculo numérico e elas não funcionaram bem em algumas tarefas que as CPUs são naturalmente projetadas para funcionar, esperando-se assim que a maioria das aplicações usará tanto CPUs quanto GPUs, executando as partes sequenciais na CPU e as partes numericamente intensivas nas GPUs. Por isso que o modelo de programação CUDA (Compute Unified Device Architecture), introduzido pela NVIDIA em 2007, foi projetado para admitir a execução conjunta de CPU / GPU de uma aplicação.



É importante resaltar que o desempenho não é o único fator de decisão quando os desenvolvedores de aplicação escolhem os processadores para rodar suas aplicações. O mais importante refere-se a presença do processador no mercado. O motivo é simples, o custo do desenvolvimento de software é melhor justificado por uma população de clientes muito grande. Há pouco tempo atrás, apenas aplicações de elite, patrocinadas pelo governo e grandes corporações, tiveram sucesso em sistemas de computação paralela tradicionais.

O domínio dessa arquitetura como ferramenta é um dos pontos altos desse projeto, uma vez que todos os avanços científicos e tecnológicos esperados estão intimamente ligados a essa arquitetura.

### **Modelagem Atmosférica**

A manufatura digital abrange as etapas de planejamento e projeto, os processos de detalhamento e validação, o desenvolvimento de recursos de modelagem e simulação e por fim, a extração dos dados de manufatura e instruções de trabalho. É evidente a necessidade de interação entre distintas áreas do conhecimento para compor o trabalho uma vez que torna-se necessária a permanente interação entre o pesquisador que disponibiliza os dados e o conhecimento a ser extraído deles e o pesquisador que auxilia no reconhecimento das necessidades computacionais. Esta constante interação comprova o fortalecimento do aspecto multidisciplinar da proposta.

Foi escolhido como alvo de estudo um modelo numérico de tempo, a descrição desse modelo torna-se necessária, uma vez que ele é utilizado como referência inicial na construção deste trabalho.

O Weather Research and Forecasting (WRF) é um modelo numérico de previsão do tempo e simulação atmosférica desenvolvido para pesquisa e aplicações operacionais. Este sistema foi desenvolvido para constituir a próxima geração de modelos de assimilação de dados e previsão de mesoescala. Possui sistema de núcleos dinâmicos e uma arquitetura de software que permite paralelismo computacional.

O modelo é mantido e suportado como um “modelo comunitário de mesoescala”, justamente para facilitar a ampla utilização em pesquisa, especialmente na comunidade universitária. O WRF foi desenvolvido em conjunto com o NCAR, Meso-scale and Micro-scale Meteorology (MMM), o National Oceanic and Atmospheric Administration (NOAA), National Centers for Environmental Prediction (NCEP), Earth System Research Laboratory (ESRL), o Department of Defense’s Air Force Weather Agency (AFWA), o Naval Research Laboratory (NRL), o Center for Analysis and Prediction of Storms (CAPS) e a University of Oklahoma, and the Federal Aviation Administration (FAA).

O WRF possui várias opções para soluções da física e da dinâmica da atmosfera.

A framework do WRF está definido em cinco camadas:

- Camada de driver;
- Camada de mediação;
- Camada de modelo;
- Utilitário de metaprogramação chamada de registro;
- Application Programming Interface (API) para comunicação de entrada e saída.

A camada de driver trata a alocação em tempo de execução e a decomposição paralela das estruturas de dados dos domínios do modelo, bem como sua organização, interação, gestão e controle sobre os domínios aninhados, incluindo o laço do tempo principal do modelo. Cada domínio é representado como um único objeto, cada objeto desse conjunto de domínios aninhados é chamado de *nest* e são representados como uma árvore de domínios com a raiz no nível mais alto. A cada passo de tempo do modelo, envolve uma chamada recursiva dos níveis mais internos até os mais externos desta árvore, avançando cada nó e seus filhos para o próximo passo de tempo do modelo, como mostra a figura 8.

A camada de mediação possui um conjunto completo de rotinas para comunicação da camada de modelo, incluindo a interação entre processadores e

múltiplas *threads*. O WRF utilizado atualmente suporta o uso de *threads*, MPI e paralelismo de memória compartilhada utilizando OpenMP, essa pesquisa propõe o uso de programação CUDA para paralelismo baseado em GPU, criando assim uma programação híbrida, tentando utilizar o melhor dos dois mundos, o de programação para processamento serial distribuído e o para execução vetorial.

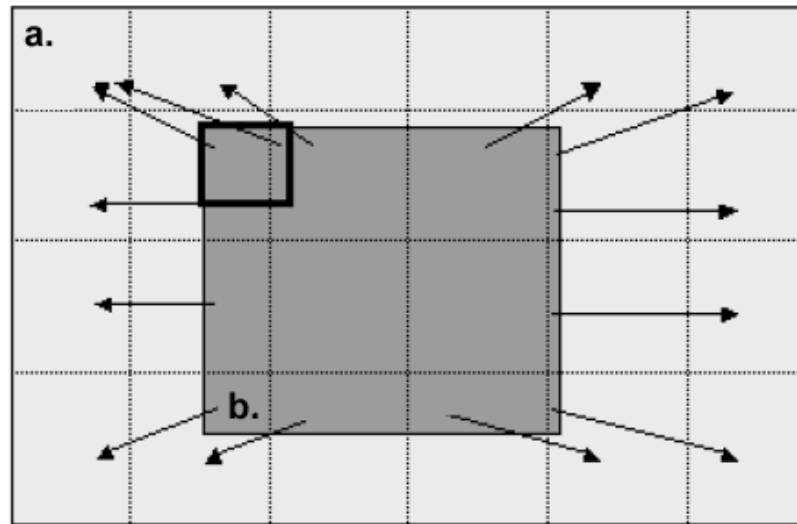


Figura 8: Representação do aninhamento entre domínios: a) domínio pai com menor resolução; b) domínio filho (nest) com maior resolução.

A camada de modelo compreende as rotinas computacionais reais que compõem o modelo: difusão, advecção, parametrizações físicas e assim por diante. As sub-rotinas são chamadas através de uma interface padrão, essas chamadas não incluem tratamento de entrada e saída de dados, controle de múltiplas *threads* ou comunicação entre processadores. Esse tipo de arquitetura garante que qualquer pacote incorporado a essa camada irá funcionar em qualquer tipo de computador paralelo cujo framework tenha suporte. As rotinas desta camada que possuem dependência de dados utilizam a camada de mediação para realizar a comunicação entre processadores. É na camada de modelo que encontram-se as diferentes opções de uso de microfísica de nuvem, dentre elas a WSM5. Seu funcionamento e relevância nesse estudo ficará mais clara nos capítulos subsequentes.

O registro trata-se de um conciso banco de dados de informações sobre a estrutura de dados do WRF. A base de dados do registro é uma coleção de tabelas que

listam e descrevem variáveis de estado e matrizes do WRF com seus atributos tais como: dimensionalidade, número de níveis do modelo, etc.. A partir deste banco de dados, o registro gera o código para as interfaces entre as camadas.

A camada de APIs compreende o conjunto de rotinas que fazem uso de bibliotecas externas tais como: NetCDF, Hierarchical Format (HDF), GRIdded Binary or General Regularly-distributed Information in Binary form (GRIB2), OpenMP, MPI, etc.. Essa camada possibilita ao WRF adicionar novos pacotes de forma independente a fim de trabalhar com diferentes formatos de dados ou diretivas de comunicação entre processadores.

## Microfísica de Nuvens

A microfísica de nuvens é crucial, mas uma parte computacionalmente intensiva do WRF. O esquema *Single Moment 5-class* (Momento simples classe 5, ou WSM-5) representa vários tipos de precipitação, condensação e efeitos termodinâmicos de liberação de calor latente. O esquema WSM5 prevê cinco categorias de hidrometria: vapor de água, água na nuvem, gelo na nuvem, chuva e neve. A **figura 9** retrata as diferentes categorias que são explicadas na tabela 1. WSM5 permite a existência de água super-resfriada e o derretimento gradual de neve.

A condensação de vapor de água para água de nuvem é expressa por

$$PCOND = (q - q_{sw})[\Delta t(1 + \frac{L_v^2}{C_{pm}R_vT^2})]$$

onde água de nuvem evapora se  $q < q_{sw}$ ,  $\Delta t$  é a variação do tempo,  $L_v$  é o calor latente de condensação,  $C_{pm}$  é o calor específico à pressão atmosférica constante,  $R_v$  é constantes dos gases para o ar úmido e  $T$  é temperatura.

Quando o ar está supersaturado, com relação ao gelo, a taxa de crescimento de cristais de gelo por deposição de vapor de água é:

$$PIDEP = \frac{4D_I(S_I - 1)N_I}{(A_I + B_I)}$$

onde  $D_I$  é o diâmetro do gelo de nuvem,  $S_I$  é a taxa de saturação do gelo,  $A_I$  e  $B_I$  são os termos termodinâmicos.

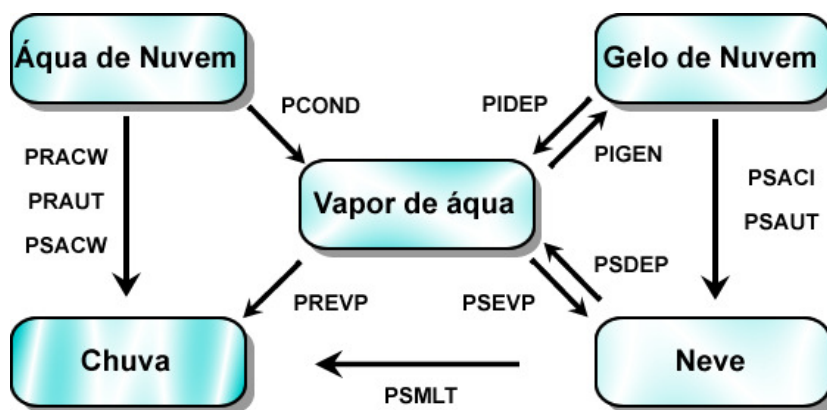


Figura 9: Fluxograma dos processos do esquema WSM5.

Tabela 1: Legenda da figura 9.

LEGENDA	DESCRIÇÃO
$P_{COND}$	Taxa de produção de condensação do vapor de água em água de nuvem.
$P_{IDEP}$	Taxa de produção de cristais de gelo por deposição de vapor.
$P_{IGEN}$	Taxa de produção de cristais de gelo por deposição de vapor (se $T < T_o$ )
$P_{RACW}$	Taxa de produção de gotas pela acreção de água de nuvem.
$P_{RAUT}$	Taxa de produção para autoconversão de água de nuvem para formar chuva.
$P_{REVP}$	Taxa de produção de evaporação da chuva.
$P_{SACI}$	Taxa de produção por acreção de gelo na nuvem por neve.
$P_{SACW}$	Taxa de produção por acreção de água de nuvem por neve.
$P_{SAUT}$	Taxa de produção por autoconversão do gelo de nuvem para formar neve.
$P_{SDEP}$	Taxa de produção por crescimento por deposição de neve.
$P_{SEVP}$	Taxa de produção pela evaporação por derretimento de gelo.
$P_{SMLT}$	Taxa de produção da água da nuvem por derretimento de neve.

Quando a temperatura  $T$  é menor que a temperatura de referência  $T_0$  e o ar esta supersaturado, em relação ao gelo, o início da tava de gelo de nuvem para vapor de água é:

$$PIGEN = \min\left[\frac{q_{I0} - q_I}{\Delta_t t}, \frac{q - q_{SI}}{\Delta t}\right]$$

onde  $q_{I0}$  é a razão de mistura dos núcleos de gelo,  $q_I$  é a razão de mistura dos cristais de gelo,  $q$  é a razão de mistura de vapor de água e  $q_{SI}$  é o valor de saturação em relação ao gelo.

A taxa de crescimento da gota de chuva por acréscimo de água de nuvem é:

$$PRACW = \frac{\pi \alpha_R E_{RC} n_{0R} q_c \Gamma(3 + b)}{4 \lambda_r^{3+b}} \left(\frac{\rho_0}{\rho}\right)^{\frac{1}{2}}$$

onde  $\alpha_R$  é 841,9 m/s,  $E_{RC} = 1$  é a eficiência da coleta da chuva,  $n_{0R} = 8 \cdot 10^6$  parâmetro de interceptação da chuva,  $q_c$  é a razão de mistura de água de nuvem,  $\Gamma$  é a função gama,  $\lambda_r$  é um declive do tamanho da distribuição de chuva,  $\rho_0$  é a densidade do ar no estado de referência, e  $\rho$  é a densidade do ar.

A taxa de produção para autoconversão de água de nuvem para formar chuva é:

$$PRAUT = \frac{0,104 g E_c \rho_0^{\frac{4}{3}}}{\mu (N_c \rho_w)^{\frac{1}{3}}} \cdot q_c^{73} H(q_c - q_{c0})$$

onde  $H()$  é a função de passo Heaviside,  $g = 9,8 \text{ m/s}^2$  a aceleração da gravidade,  $E_c$  é a eficiência de coleção,  $\mu = 1,7 \cdot 10^{-5}$  é a viscosidade dinâmica do ar,  $N_c$  é a concentração de gotículas de água na nuvem,  $\rho_w$  é a densidade da água,  $q_c$  é a razão de água de nuvem e  $q_{c0}$  é a razão de mistura crítica de água de nuvem.

A taxa de evaporação para chuva:

$$PREVP = \frac{2\pi N_{0R} (S_w - 1)}{(A_w + B_w)} \cdot \left[ \frac{0,78}{\lambda_r^2} + \frac{0,31 \Gamma \frac{(b_r+5)}{2} a^{\frac{1}{2}}}{\lambda_r^{-\frac{(b_r+5)}{2}}} \left(\frac{\mu_k}{D}\right) \left(\frac{a_R \rho}{\mu_k}\right)^{\frac{1}{2}} \left(\frac{\rho_0}{\rho}\right)^{\frac{1}{4}} \right]$$

onde  $S_w$  é a taxa de saturação da água,  $A_w$  e  $B_w$  são termos termodinâmicos,  $a_R = 841,9$  e  $\mu_k$  a viscosidade cinemática.

Taxa de acréscimo de neve através da acreção de gelo de nuvem:

$$PSACI = \frac{\pi q_I E_{SI} n_{os} |V_S - V_I|}{4} \cdot [\lambda_s^{-3} + 2D_I \lambda_s^{-2} + D_I^2 \lambda_s^{-1}]$$

onde  $E_{SI}$  é a eficiência de coleta de neve pelo gelo de nuvem,  $V_S$  é a massa ponderada da velocidade de queda da neve,  $V_I$  é massa ponderada da velocidade de queda de gelo de nuvem,  $\lambda_s$  é um declive de distribuição de tamanho de neve e  $D_I$  é o diâmetro do gelo de nuvem.

A taxa de acréscimo de água de chuva por neve:

$$PSACW = \frac{\pi a_s E_{SC} n_{os} q_c \Gamma(3 + b_s)}{4 \lambda_s^{3+b_s}} \left(\frac{\rho_o}{\rho}\right)^{\frac{1}{2}}$$

onde  $E_{SC} = 1$  é a eficiência de coleção de neve da água de nuvem,  $b_s = 0,41$ . Quando  $T > T_0$ ,  $PSACW$  irá contribuir para formar gotas de chuva.

A taxa de autoconversão de cristais de gelo para formar neve:

$$PSAUT = 1 \times 10^{-3} e^{0,025T_c} (q_i - q_{Io})$$

A taxa de crescimento de deposição de neve:

$$PSDEP = \frac{4n_{os}(S_i - 1)}{A_I + B_I} \cdot [0,65\lambda_s^{-2} + 0,44\Gamma \frac{\Gamma\left(\frac{b_s+5}{2}\right)}{\lambda_s^{\frac{b_s+5}{2}}} \cdot \left(\frac{\mu_k}{D}\right)^{\frac{1}{3}} \left(\frac{a_s \rho}{\mu_k}\right)^{\frac{1}{4}} \left(\frac{\rho_o}{\rho}\right)^{\frac{1}{4}} \frac{\Gamma\left(\frac{b_s+5}{2}\right)}{\lambda_s^{\frac{b_s+5}{2}}}]$$

onde  $A_I$  e  $B_I$  são termos termodinâmicos.

Evaporação do derretimento da neve:

$$PSEVP = \frac{4n_{os}(S_i - 1)}{A_W + B_W} \cdot [0,65\lambda_s^{-2} + 0,44\Gamma \frac{\Gamma\left(\frac{b_s+5}{2}\right)}{\lambda_s^{\frac{b_s+5}{2}}} \cdot \left(\frac{\mu_k}{D}\right)^{\frac{1}{3}} \left(\frac{a_s \rho}{\mu_k}\right)^{\frac{1}{4}} \left(\frac{\rho_o}{\rho}\right)^{\frac{1}{4}} \frac{\Gamma\left(\frac{b_s+5}{2}\right)}{\lambda_s^{\frac{b_s+5}{2}}}]$$

Taxa de produção de água de nuvem do gelo derretido:

$$PSMLT = \frac{2\pi n_{os}}{L_f} K_u (T - T_0) [0,65\lambda_s^{-2} + 0,44\Gamma \frac{\Gamma\left(\frac{b_s+5}{2}\right)}{\lambda_s^{\frac{b_s+5}{2}}}]$$

onde  $L_f$  é o calor latente de fusão,  $K_u$  representa a condutividade térmica do ar.

## CAPÍTULO 3: METODOLOGIA

### Análise de Erros

Medidas estatísticas como erro médio, raiz quadrática do erro médio e desvio padrão podem ser usadas para avaliar a qualidade das previsões do modelo. O erro médio (EM) calcula a diferença direta entre o dado observado e o dado modelado, expressando o viés do modelo para determinada variável. Se  $EM > 0$ , o modelo está superestimando a variável, em caso contrário, está subestimando.

Pode-se afirmar que quando EM tende a zero, o modelo está apresentando boa acurácia em sua predição, onde:  $N$  é o número total de dados,  $P_i$  é o dado simulado e  $O_i$  é o dado observado.

$$EM = \frac{1}{N} \sum_{i=1}^N (P_i - O_i)$$

A raiz do Erro Médio Quadrático (RSME) é a medida do quadrado da diferença entre os valores previstos e os observados. Como eleva ao quadrado a diferença, é mais sensível a erros, e por isso, definida como uma medida de precisão. O valor zero indica uma predição perfeita e esse valor aumenta conforme aumenta a diferença entre valores de previsão e observação (Wilks, 1995).

$$RSME = \sqrt{\frac{1}{N} \sum_{i=0}^N (P_i - O_i)^2}$$

### Análise de Complexidade de Algoritmos

Uma das etapas do processo de solução de problemas utilizando algoritmos é a análise dos mesmos, permitindo de forma objetiva escolher um dentre vários algoritmos disponíveis para uma mesma aplicação. Uma avaliação do custo de diferentes algoritmos possibilita a comparação e a escolha do mais adequado para resolver determinado tipo de problema.

Quando determinado o menor custo possível para resolver um problema de uma determinada classe, como o caso de multiplicação de vetores, medida da dificuldade



inerente para resolver o problema é encontrada. Quando o custo de um algoritmo é igual ao menor custo possível, então pode-se concluir que o algoritmo é ótimo.

Para medir o custo de execução de um algoritmo, podemos definir-se uma função de complexidade  $f$ , onde  $f_t(n)$  é a medida de tempo  $t$  necessário para executar um algoritmo para um problema de tamanho  $n$ . Logo, neste caso,  $f_t$  é reconhecida como a função de complexidade de tempo. É importante ressaltar que a complexidade de tempo na realidade não representa tempo diretamente, mas sim o número de vezes que determinada operação considerada relevante é executada. Se  $f_m(n)$  é uma medida de quantidade de memória  $m$  necessária para executar um algoritmo de tamanho  $n$ , então  $f_m$  é chamada função de complexidade de espaço.

A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados. Por isso, é comum considerar-se o tempo de execução de um programa como uma função do tamanho da entrada.

A análise de complexidade é realizada aplicando a definição de três cenários: melhor caso, pior caso e por fim, caso médio. O melhor caso corresponde ao menor tempo de execução sobre todas as entradas de tamanho  $n$ . O pior caso corresponde ao maior tempo de execução sobre todas as entradas de tamanho  $n$ . O custo de aplicar o algoritmo nunca é maior que  $f(n)$  baseada na análise do pior caso.

O caso médio corresponde à média dos tempos de execução de todas as entradas de tamanho  $n$ . Na análise do caso médio, uma distribuição de probabilidades sobre o conjunto de tamanho  $n$  é suposta e o custo médio é obtido com base nesta distribuição. (Cormen, 2009). Por essa razão, a análise do caso médio é geralmente mais difícil de obter do que as análises de melhor e do pior caso.

Na prática, para valores pequenos de  $n$ , qualquer algoritmo custa pouco para ser executado, mesmo aqueles ineficientes. Logo, a análise de algoritmos torna-se interessante para valores grandes de  $n$ . Para tal, considera-se o comportamento assintótico das funções de custo. O comportamento assintótico de  $f(n)$  representa o limite do comportamento do custo quando  $n$  cresce. A análise do algoritmo geralmente conta com apenas algumas operações elementares e, em muitos casos, somente uma operação elementar. A medida de complexidade relata o crescimento assintótico da

operação considerada. A definição seguinte relaciona o comportamento assintótico de duas funções distintas.

Definição: Uma função  $f(n)$  domina assintoticamente outra função  $g(n)$  se existem duas constantes positivas  $c$  e  $m$  tais que se  $n \geq m$ , então  $|g(n)| \leq c|f(n)|$ .

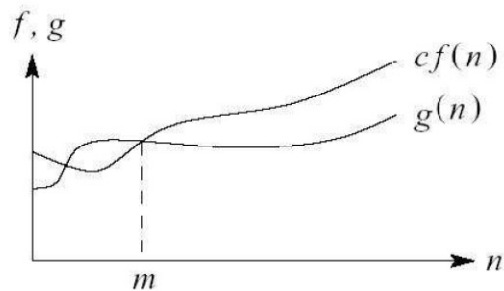


Figura 10: Dominação assintótica de  $f(n)$  sobre  $g(n)$ . Fonte: CORMEN et al., 2009.

Knuth (1976) sugeriu uma notação para domínio assintótico. Para expressar que  $f(n)$  domina assintoticamente  $g(n)$  escreve-se  $g(n) = O(f(n))$ . A Figura 10 mostra um exemplo gráfico de dominação assintótica que ilustra a notação  $O$ . As funções de complexidade de tempo são definidas sobre os inteiros não negativos, ainda que possam ser não inteiros.

Definição da notação  $O$ : Uma função  $g(n)$  é  $O(f(n))$  se existem duas constantes positivas  $c$  e  $m$  tais que  $g(n) \leq cf(n)$ , para todo  $n \geq m$ . Por exemplo, seja  $g(n) = (n+1)^2$ , logo  $g(n)$  é  $O(n^2)$ , quando  $m=1$  e  $c=4$ , porque  $(n+1)^2 \leq 4n^2$  para  $n \geq 1$ .

A análise de complexidade de algoritmos é uma ciência rica em elementos e seu estudo se estende fazendo uso de outras técnicas que neste trabalho não serão evidenciadas.

## Microfísica de Nuvem

Para o referente estudo, escolheu-se estudar um dos muitos componentes do modelo numérico de previsão de tempo WRF para demonstrar como o uso de um novo paradigma computacional poderia influenciar o desempenho de programas que executam de forma intensiva operações matemáticas. Por ter sua estrutura separada em módulos e seus scripts sendo utilizados como rotinas padronizadas, bastaria

reprogramar uma dessas bibliotecas para supostamente obter melhores resultados temporais.

Como mencionado, rotinas que envolvem a parametrização de microfísica de nuvem possuem alto custo computacional, bastava agora definir qual seria a microfísica. Foram realizados estudos comparativos para a região utilizando diferentes microfísicas. Para fins de comparação, foram definidos parâmetros idênticos para todas rodadas, onde a única variável alterada era a que definia a microfísica a ser usada. Ao todo, foram testadas doze microfísicas de nuvens. Comparativos temporais e qualitativos foram realizados e dentre todas as microfísicas, foram selecionadas três. Todos os testes realizados nessa fase foram regidos por um script, previamente programado para executar rodada após rodada fixando todos os parâmetros, com exceção da microfísica de nuvem.

O próximo fator a ser considerado foi o período de análise. A importância desse parâmetro de entrada é percebido uma vez que, para solicitar computação a microfísica de nuvem, é necessário haver chuva de microfísica. Após alguns testes, foi verificado um período em que esse fenômeno era bem definido e observado. A alta variação de custo temporal, por si só, foi um indicativo do custo computacional necessário para processar a microfísica, contudo, foram analisadas as saídas do modelo e verificadas as quantidades de chuvas convectivas e não convectivas, definindo assim o período de análise.

Como condições iniciais e de fronteira do WRF foram utilizados os dados FNL-NCEP, com resolução horizontal de  $1^\circ \times 1^\circ$  de longitude e latitude, 27 níveis verticais e resolução temporal de 6 horas com data inicial em 0 UTC do dia 16 de setembro de 2012 e término em 20 de setembro de 2012 às 0 UTC, totalizando 96 horas. Foi utilizada uma única grade com 200 x 200 pontos e resolução horizontal 5 quilômetros e 27 níveis na vertical.

Para a execução dos testes, foi criado um script que executava de forma automática uma série restrita de rodadas baseada em uma lista previamente definida pelo usuário. O programa prepara os dados na parte de pré-processamento do modelo, conhecida como WPS, envia a referência para o kernel do modelo, conhecido como WRF, nesse local, foi adicionado um arquivo namelist.template, esse arquivo possui

flags que são trocadas conforme a necessidade de cada rodada, basicamente, troca-se a microfísica de nuvens. Todos os resultados de saída são salvos com nomes diferentes para fins de análise e comparação. O esquema da figura 11 demonstra um pequeno fluxograma de como ocorre esse processo.

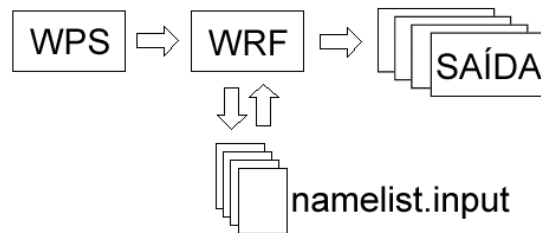


Figura 11; Fluxograma da execução dos testes de microfísica,

Após o término do processo de definição de período, ocorreu a análise comparativa entre os dados de saída do modelo e os dados observados. Para essa análise, foram consultadas trinta e quatro estações, referenciadas na tabela 2.

Tabela 2: Lista de estações de observações usadas no estudo comparativo entre diferentes microfísicas.

Número	Estação	Localização		
		latitude	longitude	altitude (m)
1	CHUI	-33,7418	-53,6714	26,00
2	RIO GRANDE	-32,0333	-52,4000	2,46
3	MOSTARDAS	-31,2478	-51,1057	10,00
4	JAGUARAO	-32,5500	-53,3800	47,00
5	BAGE	-31,3478	-54,0133	230
6	DOM PEDRITO	-30,9925	-54,8153	170
7	LIVRAMENTO	-30,8333	-55,6000	328
8	QUARAI	-30,3686	-56,4372	124
9	ALEGRETE	-29,7116	-55,5261	121
10	SAOGABRIEL	-30,3414	-54,3108	126

11	CACAPAVA	-30,5477	-53,4675	450
12	CANGUCU	-31,4058	-52,7011	464
13	CAMAQUA	-30,8106	-51,8347	108
14	SAOBORJA	-28,6494	-56,0156	83
15	SANTIAGO	-29,1914	-54,8856	394
16	SANTAMARIA	-29,7	-53,7	95
17	RIOPARDO	-29,8733	-52,3825	111
18	POA	-30,05	-51,1666	46,97
19	TRAMANDAI	-30,097	-50,4353	1
20	SAOLUIZG	-28,4172	-54,9625	245
21	CRUZALTA	-28,6036	-53,6736	432
22	SOLEDADE	-28,8536	-52,5417	667
23	BENTOG	-29,1672	-51,5347	640
24	CANELA	-29,3688	-50,8274	830
25	TORRES	-29,3503	-50,1331	4,5
26	SANTAROSA	-27,8901	-54,4797	276
27	SANTOAUGUSTO	-27,85	-53,7833	550
28	PALMEIRA	-27,9199	-53,3174	642
29	PASSOFUNDO	-28,2294	-52,4039	684
30	LAGOAVER	-28,2219	-51,5122	842
31	VACARIA	-28,5136	-50,8828	986
32	SAOJOSE	-28,7514	-50,0583	999,99
33	FREDERICOW	-27,3956	-53,4294	490
34	ERECHIM	-27,6603	-52,3064	765

Os dados analisados foram os de precipitação. Estes dados foram comparados aos dados de saída do modelo. Uma vez que a chuva total do modelo é definida como o somatório da chuva convectiva mais a chuva não convectiva(chuva de microfísica) no

período, foram calculadas a precipitação total para cada estação e para cada ponto do modelo referente a região onde localiza-se as estações. Com esses valores, foram calculados para cada ponto o erro médio e o erro médio quadrático, obtendo o indicativo de quais microfísicas de nuvens forneciam menores erros de previsão.

Após a aplicação dos critérios iniciais de escolha da microfísica de nuvens, foram devidamente avaliados os códigos escritos na linguagem Fortran. Não obstante das ponderações qualitativas no que tange a assertividade do modelo e das tomadas de tempo, que possuem implicações diretas no quão oneroso é a resolução computacional da microfísica de nuvens para o modelo, tomou-se principalmente como referência a existência massiva de operações do tipo SIMT sob matrizes de dados e, em segundo plano, a legibilidade, a quantidade de linhas do código e trabalhos anteriores realizados (Mielikainen et al., 2012). A microfísica de nuvens WSM5 encaixava-se perfeitamente na pesquisa e foi selecionada como a primeira a receber todas as atenções.

### **Análise do Algoritmo da Microfísica de Nuvem**

A decomposição da microfísica WSM5 começa por localizar o código na estrutura do WRF. O script encontra-se localizado na área referente física do modelo e responde pelo nome de *module\_mp\_wsm5.F77*. Com o uso de um editor de texto, foi aberto o script para apreciação. A priori, foi feita uma avaliação dos tipos predominantes de operações computacionais existentes. Foi determinado que a maioria das operações eram do tipo SIMT sobre matrizes. O código foi decodificado sob a forma de fluxogramas de dados que simplificavam o processo de avaliação uma vez que permitiam uma avaliação visual mais direta.

Aplicou-se as técnicas de modularização do algoritmo, classificando as parcelas de código em blocos. Cada bloco era definido como uma união de pequenas parcelas de código que de alguma forma possuíam uma ligação entre si. Logo, cada bloco passava a possuir uma identidade e era apresentado de forma genérica, sem muitas especificações a respeito das operações mais internas. Um exemplo de como deu-se esse processo está na figura 12.

253	do k = kts, kte	Bl. 3
254	do i = its, ite	
255	qci(i,k,1) = max(qci(i,k,1),0.0)	
256	qrs(i,k,1) = max(qrs(i,k,1),0.0)	
257	qci(i,k,2) = max(qci(i,k,2),0.0)	Bl. 4
258	qrs(i,k,2) = max(qrs(i,k,2),0.0)	
259	enddo	Bl. 5
260	enddo	
261	do k = kts, kte	
262	do i = its, ite	
263	cpm(i,k) = cpmcal(q(i,k))	Bl. 4
264	xl(i,k) = xlcal(t(i,k))	
265	enddo	Bl. 5
266	enddo	
267	loops = max(nint(delt/dtcdcr),1)	
268	dtcd = delt/loops	
269	if(delt.le.dtcdcr) dtcd = delt	

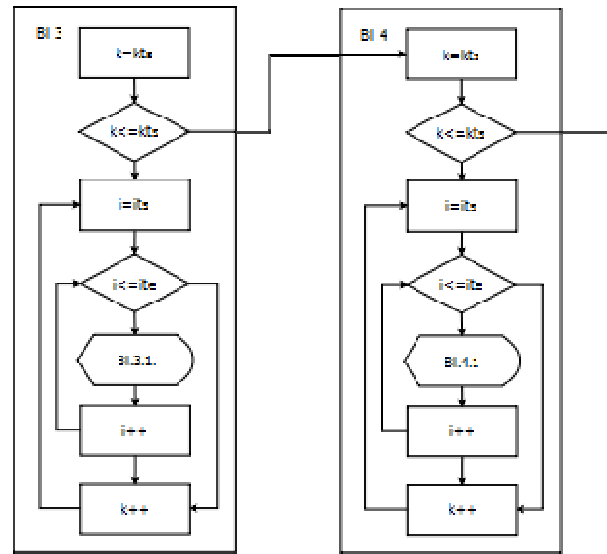


Figura 12: Demonstração da transcrição de código escrito para fluxograma de dados.

O bloco 3 foi definido como proprietário das linhas compreendidas no intervalo de 253 a 260 e o bloco 3.1 entre 255 e 258. Essa notação permite percorrer todos os níveis do código e separar os trechos para avaliação unitária. Essa técnica foi aplicada em todo o código do WSM5. Com a montagem completa do fluxograma, torna-se mais simples analisar a complexidade do algoritmo.

Todas as parcelas de código foram analisadas separadamente e, na maioria dos casos, se subdividiam em parcelas menores de código como uma estrutura em árvore, onde as folhas normalmente representavam computações de um tempo de execução. Os blocos mais internos herdavam a complexidade de seus blocos pai, fazendo assim uma avaliação da complexidade assintótica do algoritmo.

As funções do algoritmo foram analisadas separadamente e a cada uma delas foi atribuída uma complexidade e por fim a complexidade total do algoritmo quando utilizado para resolver a microfísica de nuvens.

Ao finalizar a descrição da arquitetura do algoritmo e calcular sua complexidade, foram então propostas soluções alternativas fazendo uso da tecnologia de paralelismo baseada em GPU. A cada bloco de código definido como potencialmente paralelizável, foi proposta uma solução escrita em CUDA e posteriormente recalculada a sua

complexidade. A intenção era tornar um trecho de código de complexidade  $O(n)$  ou superior, em  $O(1)$ , como mostra a figura 13, onde um trecho de código de complexidade  $O(n)$ , ao ser transcrito para CUDA, passa a ter ser  $O(1)$ .

Diagrama de laço de repetição de um nível(N1).

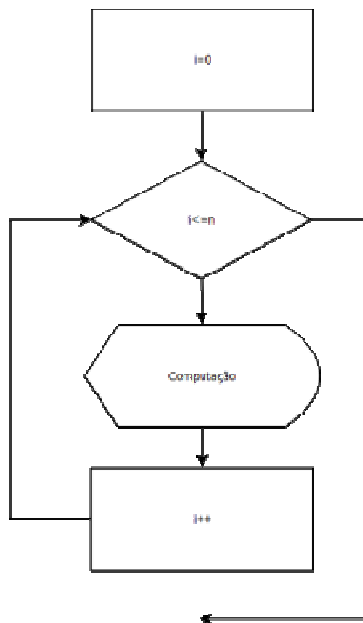


Diagrama de laço de repetição em uma arquitetura CUDA (C1).

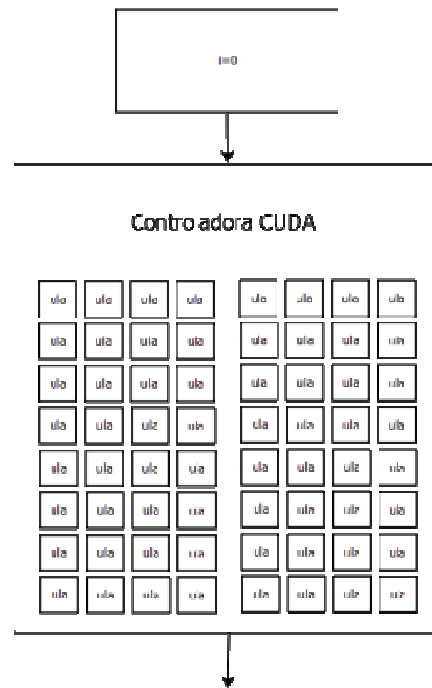


Figura 13: Trecho de código  $O(n)$  para  $O(1)$ .

### O ambiente para testes de performance

O código fonte do WRF é organizado em uma estrutura de pastas e subpastas cada uma delas contendo parcelas do modelo. Todas as microfísicas de nuvens encontram-se dentro da pasta *phys*. A técnica utilizada para verificar a validade da aplicação de paralelismo baseadas em GPU consistiu em criar cópia dos arquivos de microfísica e adicioná-los ao WRF como se fossem novas microfísicas passíveis de serem acessadas e executadas através de alteração do arquivo de configuração *namelist.input* alterando o parâmetro de microfísica.



A criação de cinco scripts adicionais reflete o fato de pequenas alterações no código fonte poderem denegrir ou melhorar o desempenho temporal do modelo, sendo assim, cria-se uma série pré definida de testes com a intenção de avaliar qual parâmetro obtem o melhor resultado.

Os cinco scripts precisam ser adicionados a estrutura do WRF e serem passíveis de reconhecimento do programa. Devido a natureza do instalador do WRF, torna-se necessária a compilação em separado. Isso acontece porque o WRF primeiramente constroi seus modulos um a um, e no caso da microfísica, ele possui um procedimento que consiste em abrir todos os scripts escritos sob o formato fortran 77, retirar todas as linhas de comentários e em branco e salvar os novos scripts como fortran 90. Dessa forma, torna-se necessaria uma intervenção a fim de impedir que ele retire as *flags* OpenACC que sempre começam com “!” (símbolo que indica linha comentada em Fortran).

Foi criado um script específico para a compilação e por fim, composição do driver das microfísicas. Basicamente o script compila os módulos separadamente com os parâmetros requeridos. Torna-se necessário ainda a adição do parâmetro “-acc”, que indica o uso das diretivas do OpenACC na composição do executável do modelo, no arquivo de configuração do instalador, *configure.wrf*. Feito isso, pode-se executar o instalador normalmente.

Para testar os novos algoritmos para a microfísica wsm5, basta modificar os parâmetros de entrada do modelo como citado anteriormente. Este procedimento facilita a execução automática e otimiza o tempo necessário para realizar testes com pequenas variações no código.

### **Aplicando técnicas de Paralelismo OpenACC**

Existem algumas maneiras de aplicar técnicas de paralelismo disponíveis no OpenACC, todas passam pela definição da região de interesse a ser paralelizada. Tudo que ficar interno às regiões definidas é executado na GPU (*device*) e tudo que ficar externo às regiões é executado na CPU (*host*). As principais diretivas são “!*\$acc*

*kernels*” e “*!\$acc parallel*” que definem o início de uma ou mais regiões a serem executadas no *device* em paralelo.

Definir uma região de *kernel* é mais implícito, dando ao compilador a liberdade de encontrar o mapa de paralelismo de acordo com o requerimento do código alvo de paralelismo (Wolfe, 2012). A construção de uma região do tipo *parallel* é mais explícita e requer uma análise mais profunda do programador para definir se é apropriado. O comportamento de cada uma dessas escolhas pode depender do compilador que se está usando.

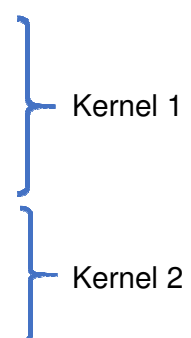
Ao construir uma região do tipo *kernel*, a API do OpenACC automatiza o processo de paralelismo inclusive definindo novas regiões de *kernel*. Como apresentado no trecho de código abaixo:

```
!$acc kernels

do i=1,n
  a(i) = 0.0
  b(i) = 1.0
  c(i) = 2.0
end do

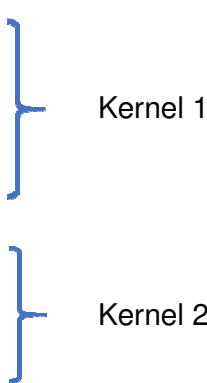
do i=1,n
  a(i) = b(i) + c(i)
end do

!$acc end kernels
```



Cada uma das regiões irá executar seus trechos de código como duas regiões paralelas independentes. O equivalente na construção de uma região do tipo *parallel* seria:

```
!$acc parallel
do i=1,n
  a(i) = 0.0
  b(i) = 1.0
  c(i) = 2.0
end do
!$acc end parallel
!$acc parallel
do i=1,n
  a(i) = b(i) + c(i)
end do
!$acc end parallel
```



Neste caso, cada região a ser executada em paralelo no *device* é definida pelo programador de forma mais explícita. Algumas outras diretivas são relevantes nesse contexto e serão evidenciadas na tabela 3.

Tabela 3: Lista de directivas OpenACC.

<code>copy ( list )</code>	Aloca memória na GPU e copia o dado do <i>host</i> para o <i>device</i> e após realizado o processamento, do <i>device</i> para o <i>host</i> .
<code>copyin ( list )</code>	Aloca memória na GPU e copia o dado do <i>host</i> para o <i>device</i> .
<code>copyout ( list )</code>	Aloca memória na GPU e copia o dado do <i>device</i> para o <i>host</i> .
<code>create ( list )</code>	Aloca memória na GPU, mas não copia.
<code>present ( list )</code>	Quando o dado já está presente na GPU ou outra região que contenha o dado.
<code>present_or_coyin (list)</code>	Testa a presença do dado na GPU, em caso negativo, copia do <i>host</i> para o <i>device</i> .
<code>present_or_copyout (list)</code>	Testa a presença do dado na GPU, em caso negativo, copia do <i>device</i> para o <i>host</i> .

Estas diretivas são importantes uma vez que definem a maneira como transitam os dados entre o *host* e o *device*, minimizando assim o tempo gasto para transferências de dados entre eles.

## CAPÍTULO 4: RESULTADOS

### Período de Análise

Um dos fatores determinantes para análise de depreciação de desempenho na execução de um modelo numérico de tempo passa pela escolha de determinada situação em que verifica-se a ocorrência do fenômeno a ser estudado, no caso, a microfísica de nuvem. Para que as funções que calculam microfísica de nuvem sejam utilizadas, é necessária a ocorrência da chamada chuva de microfísica.

Do período selecionado, pode-se observar a ocorrência de chuvas, conforme a figura 14, verifica-se a ocorrência de precipitação nas estações no período de 16 de setembro de 2012 a 20 de setembro de 2012. Cada ponto representa a localização da estação e a coloração representa a quantidade de precipitação total observada.

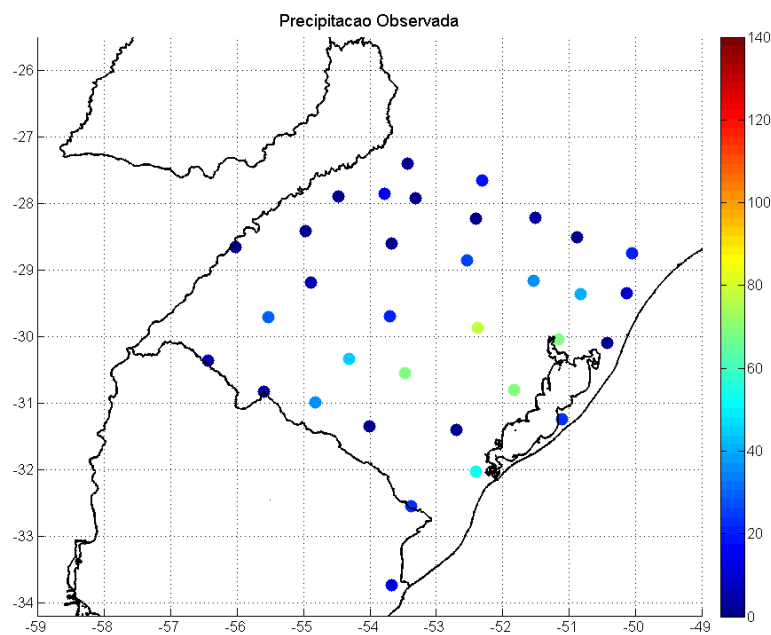


Figura 14: Precipitação total observada nas estações meteorológicas no período entre 16 de setembro de 2012 a 20 de setembro de 2012.

Para a definição de qual microfísica seria escolhida para a continuidade do trabalho, foram analisadas todas as saídas de dados oriundas do modelo WRF. As microfísicas 4, 8 e 10 foram escolhidas para esta fase do processo de pesquisa por

apresentarem bons resultados em trabalhos de análise de dados realizados para a região.(Gomes et al. 2012 e Gambetá, 2013)

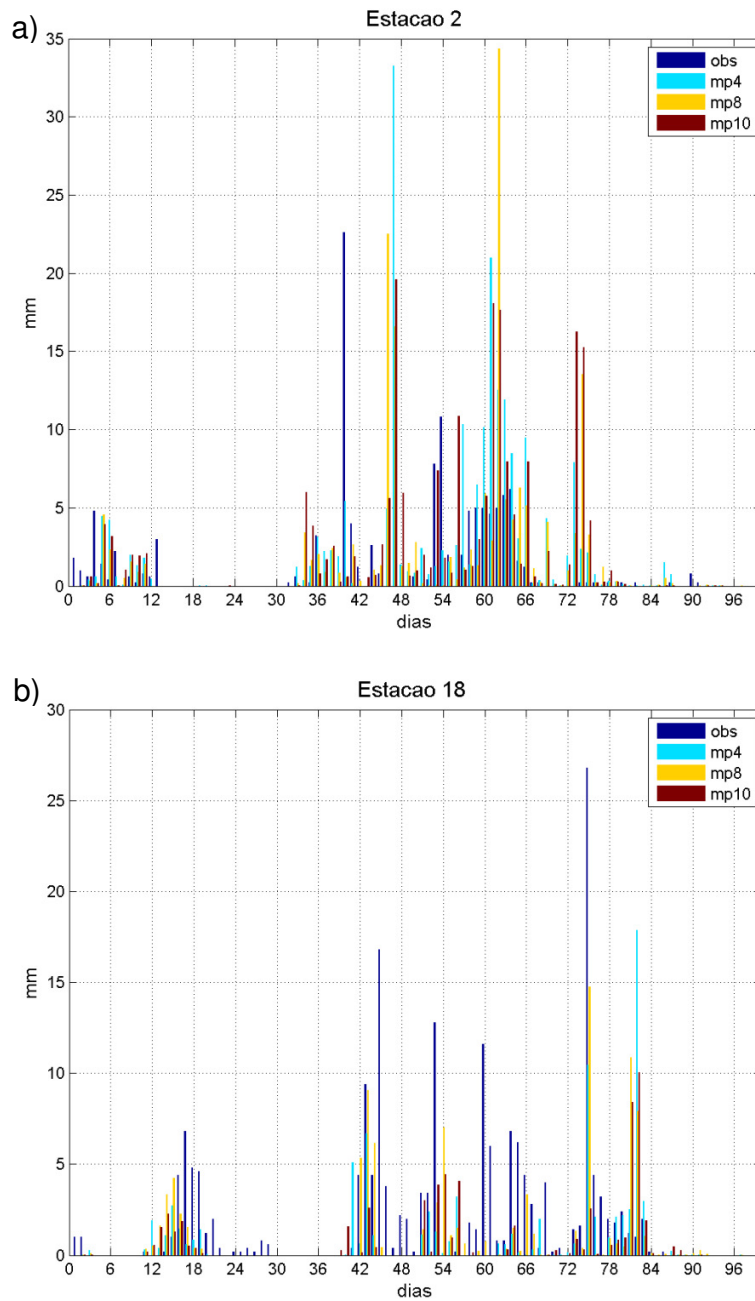


Figura 15: Dados observados e dados oriundos das saídas do modelo WRF com o uso de diferentes microfísicas de nuvem, a) Estação Meteorológica de Rio Grande.b) Estação Meteorológica de Porto Alegre.

As figuras subsequentes mostram a precipitação observada e a estimada por cada microfísica no modelo. Foram escolhidas a estação meteorológica de Rio Grande e mais quatro estações do entorno para apreciação, contudo, o estudo comparou os dados observados em todas as estações com a saída do modelo.

Tornou-se necessária uma comparação mais elegante para verificar a qualidade do modelo e das microfísicas escolhidas e para isso, foram calculados os erros médios quadráticos para a estação de Rio Grande conforme, figura 16.

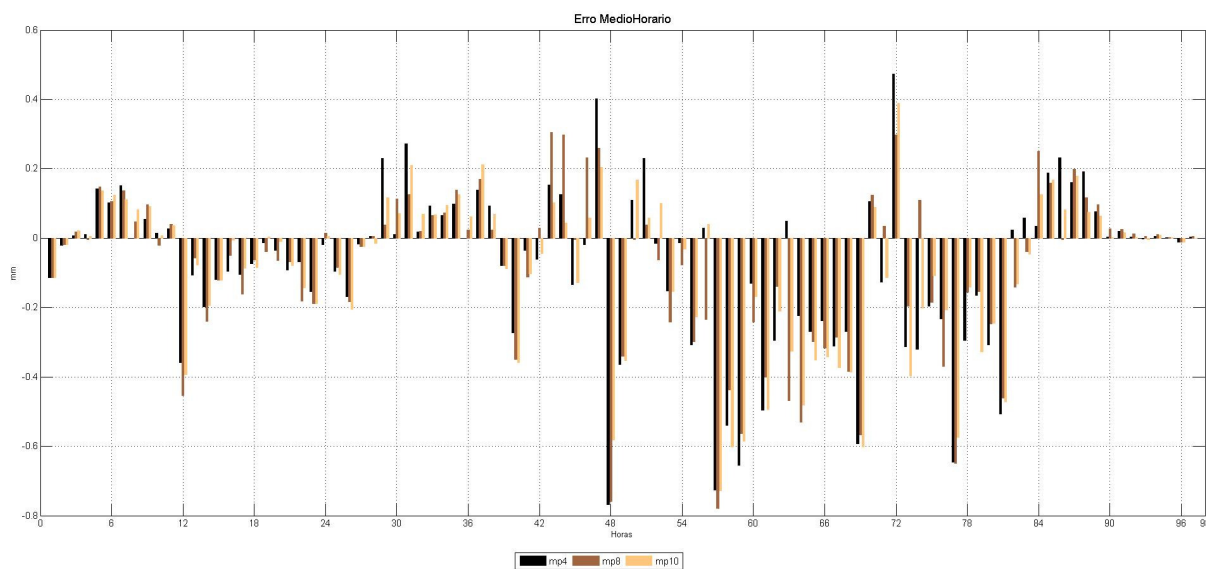


Figura 16: Erro médio, dados observados versus dados modelo WRF com microfísicas 4, 8 e 10. Estação meteorológica de Rio Grande.

As microfísicas apresentam um comportamento muito similar, normalmente acompanhando erros superiores e inferiores salvo algumas exceções. Neste momento, qualquer uma das microfísicas serviria.

Como citado, um ponto relevante na escolha da microfísica WSM5 foram os trabalhos realizados e o fato de ela apresentar bons resultados para a região, bastando apenas verificar a ocorrência de chuvas convectivas e não convectivas a fim de confirmar o uso de processos computacionais. Na figura 17-a, está demonstrado o total de chuva convectiva oriunda da saída do modelo com o uso da microfísica WSM5 e na figura 17-b a precipitação total não convectiva.

A precipitação, conferida pelo somatório das chuvas convectivas e não convectivas está relatado na figura 17.

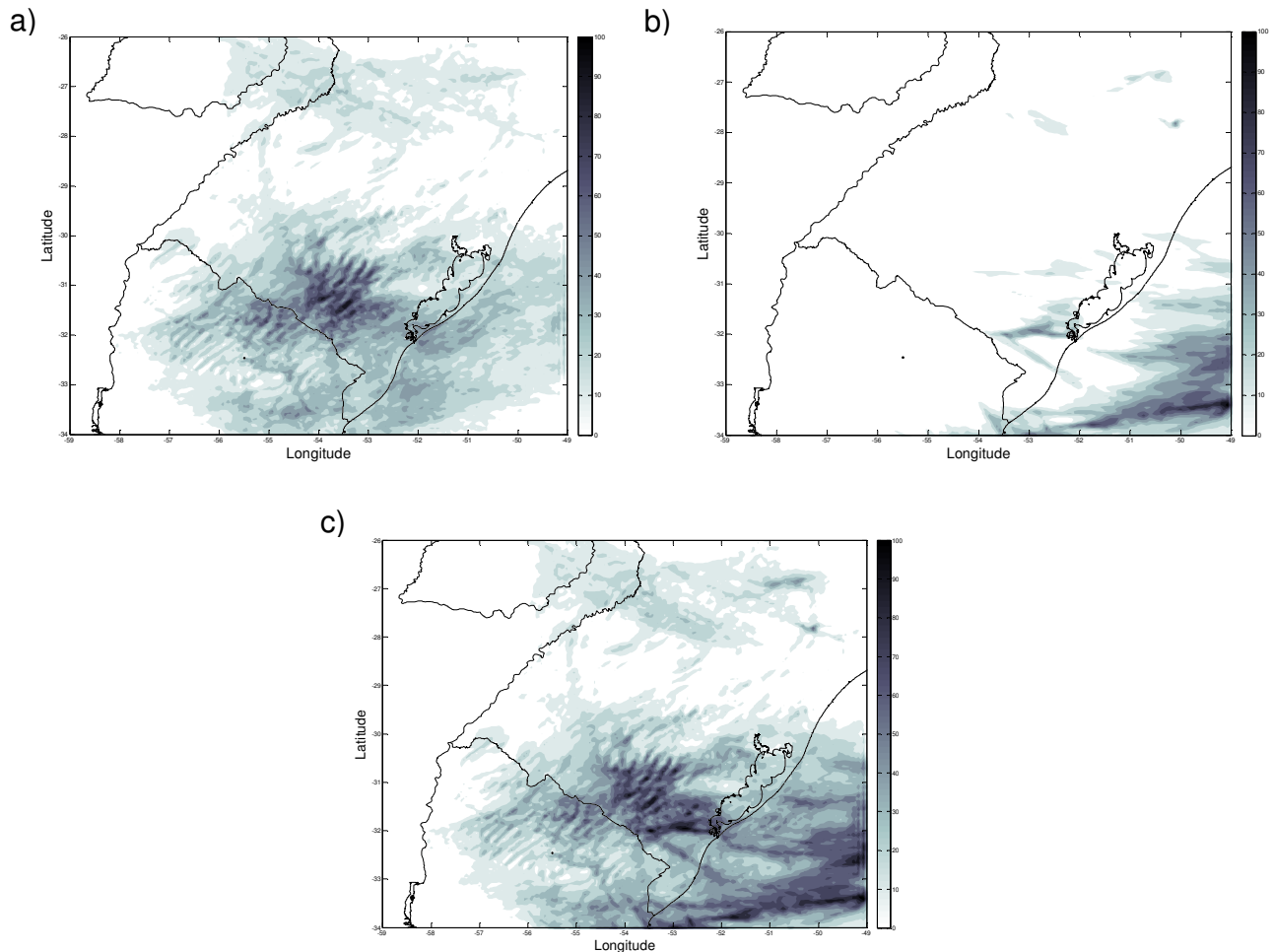


Figura 17: a) Chuva convectiva observada na saída do modelo com o uso da microfísica de nuvem WSM5. b) Chuva não convectiva observada na saída do modelo com o uso da microfísica de nuvem WSM5. c) Chuvas convectivas somadas as não convectivas observada na saída do modelo com o uso da microfísica de nuvem WSM5.

O período escolhido possui ocorrência de chuvas de microfísica, indicando assim a existência de processos relacionados a microfísica de nuvem. Contudo, para conferir o quão oneroso torna-se o processo de adição do cálculo da microfísica, tomadas de tempo foram realizadas, lembrando que os únicos fatores mutáveis são a

adição de microfísica e transformação de código paralelo CPU em código paralelo CPU + GPU.

Tabela 4: Demonstração dos resultados temporais, em segundos, da execução do modelo WRF, rodada completa, com e sem o uso da microfísica de nuvens WSM5.

	Rodada 1	Rodada 2	Rodada 3	Média
CPU sem microfísica	28529	28455	28617	28534
CPU com microfísica	58548	58485	58496	58509

A tabela 4 demonstra a tomada de tempo em segundos necessário para executar toda a rodada de teste completa sem o uso de microfísica e com o uso da microfísica WSM5. Foram executadas três rodadas com a mesma configuração de entrada e feita a média aritmética dos tempos. Percebe-se que com a adição da microfísica o tempo praticamente dobra, demonstrando que o cálculo de microfísica de nuvem exige recursos computacionais de forma extrema em relação à computação de outras partes do modelo.

O próximo passo consiste em avaliar o desempenho temporal do modelo a partir das rodadas configuradas em uma hora (tabela 5). Nesse momento executou-se o modelo com e sem a microfísica e os resultados foram similares em uma escala de tempo menor, ambos os resultados demonstram que o uso da microfísica WSM5 aumenta de 50% a 105% o tempo de execução do modelo se ocorrem chuvas de microfísica no período especificado, mostrando assim ser um bom alvo de análise para rodadas menores, otimizando assim o tempo de avaliação dos resultados de cada tentativa

Tabela 5: Demonstração dos resultados temporais, em segundos, da execução do modelo WRF, rodada de uma hora, com e sem o uso da microfísica de nuvens WSM5.

	Rodada 1	Rodada 2	Rodada 3	Média
CPU sem microfísica	307	308	307	307
CPU com microfísica	470	469	475	471



## Testes Preliminares

Foram separados trechos de código que apareciam muitas vezes no algoritmo da microfísica WSM5, essas parcelas de computação foram replicadas em um ambiente de teste com o intuito de verificar se a computação realizada na CPU era mais rápida que na GPU. Para todas as avaliações, o código foi devidamente preparado para imprimir o tempo de cada computação e a variável de ambiente `PGI_ACC_TIME` exportada como 1 para adicionar ao resultado da computação o tempo gasto na transição dos dados entre o *host* e o *device* e o tempo para computar a operação. O valor de *n* também foi incrementado com a intenção de aumentar o tamanho da entrada, verificando assim a variação temporal de uma entrada menor e uma maior.

Uma das computações predominantes era a de zerar uma matriz de dados ou outro valor qualquer.

```
Call system_clock(rtime1)
!$acc kernels
  do j=1,n
    do i=1,n
      a(i,j) = 0.0
    end do
  end do
!$acc end kernels
Call system_clock(rtime2)
Print *, ' tempo kernel: ', rtime2-rtime1

Call system_clock(rtime1)
!$acc parallel
  do j=1,n
    do i=1,n
      a(i,j) = 0.0
    end do
  end do
!$acc end parallel
Call system_clock(rtime2)
Print *, ' tempo parallel: ', rtime2-rtime1

Call system_clock(rtime1)
  do j=1,n
    do i=1,n
      c(i,j) = 0.0
    end do
  end do
Call system_clock(rtime2)
Print *, ' tempo CPU: ', rtime2-rtime1
```

Este trecho de código exemplifica o teste de duas *flags* diferentes do OpenACC e a computação serial executada na CPU utilizando o mesmo tipo de operação, mostrando assim qual a variação temporal que implica o uso de uma ou outra. Em um primeiro momento ao avaliar os resultados desta computação, o uso da *flag parallel* demonstrava um resultado muitas vezes mais rápido que seu antecessor *kernel*, contudo, ao trocar a ordem em que ocorriam no programa, o tempo da *flag parallel* aumentava e *kernel* diminuía. De acordo com (WOLFE, 2009), existe um período de latência necessário para ligar acionar a GPU e inicializá-la, esse tempo é constante e ocorre ao executar o código que faz uso da GPU. Caso o programa seja executado novamente em seguida, a placa já encontra-se em estado de ligada e não é necessário reativá-la, tornando assim a mesma execução mais rápida. Para minimizar esse dano temporal, pode ser feito uso da rotina *acc\_init()*, ao fazer isso, a placa é apenas inicializada, isolando assim qualquer custo de inicialização, do custo computacional.

Para testar o efeito temporal de diferentes arranjos de uso das diretivas OpenACC, foram propostas uma série de testes que consistiam em separar um trecho de código encontrado dentro do script da microfísica de nuvens WSM5, aplicar diferentes diretivas sobre esse código e variar o tamanho da entrada de dados. O trecho de código abaixo demonstra o uso da diretiva *kernels*, deixando assim todo o trabalho de decisão de como ocorrerá à paralelização em GPU para o compilador. Este trecho de código foi chamado de Kernel Puro.

#### KERNEL PURO

```
!$acc kernels
  do j = 1, n3
    do i = 1, n2
      x1(i,j)=k+1
      y1(i,j)= (a * (a*exp((log(a/x1(i,j))))*(a)+a*(1.- (
        a/x1(i,j)))))) / (x1(i,j) - (a*exp((log(a/x1(i,j))))*(a)
        +a*(1.-(a/x1(i,j))))))
    enddo
  enddo
!$acc end kernels
```

O próximo trecho de código fez uso da diretiva *parallel* e foi chamado de Parallel Puro.

#### PARALLEL PURO

```
!$acc parallel
  do j = 1, n3
    do i = 1, n2
      x1(i,j)=k+1
      y1(i,j)= (a * (a*exp((log(a/x1(i,j))))*(a)+a*(1.- (
a/x1(i,j)))))) / (x1(i,j) - (a*exp((log(a/x1(i,j))))*(a)
+a*(1.-(a/x1(i,j))))))
    enddo
  enddo
!$acc end parallel
```

Os dois trechos de código a seguir, nomeados como Kernel Fluxo de Dados e Parallel Fluxo de Dados, foram criados fazendo o uso do aspecto de controle de fluxo de dados entre a CPU e GPU, testando a existência do dado na GPU e em caso positivo, não requerendo a transferência ou alocação de memória do referente dado no sentido CPU-GPU. A diretiva *data* também permite definir os dados ou matrizes de dados que serão enviados a GPU, quais os dados que deverão ser criados dentro da GPU e quais dados são requisitados para retornar à CPU, minimizando assim a latência necessária para realizar transferências de dados entre GPU e CPU.

#### KERNEL FLUXO de DADOS

```
!$acc data present_or_copyin(k) present_or_create(x3(:, :))
present_or_copyout(y3(:, :))
!$acc kernels
  do j = 1, n3
    do i = 1, n2
      x1(i,j)=k+1
      y1(i,j)= (a * (a*exp((log(a/x1(i,j))))*(a)+a*(1.- (
a/x1(i,j)))))) / (x1(i,j) - (a*exp((log(a/x1(i,j))))*(a)
+a*(1.-(a/x1(i,j))))))
    enddo
  enddo
!$acc end kernels
!$acc end data
```

**PARALLEL FLUXO de DADOS**

```

!$acc data present_or_copyin(k) present_or_create(x3(:, :))
present_or_copyout(y3(:, :))
!$acc parallel
  do j = 1, n3
    do i = 1, n2
      x1(i, j)=k+1
      y1(i, j)= (a * (a*exp((log(a/x1(i, j))))*(a)+a*(1.- (
      a/x1(i, j)))))) / (x1(i, j) - (a*exp((log(a/x1(i, j))))*(a)
      +a*(1.-(a/x1(i, j))))))
    enddo
  enddo
!$acc end parallel
!$acc end data

```

O resultado do teste de sensibilidade está expresso em milisegundos na tabela 6, onde a primeira coluda representa o nome dedicado a cada trecho de código e a primeira linha o nome de cada rodada de testes, representando também o tamanho de todas as matrizes de entradas de dados. É possível verificar que com o aumento do tamanho da matriz de dados, o menor e o maior resultado temporal varia muito, com excessão das rodadas Matriz 640X640 e Matriz 6400X6400, onde a CPU já começa a apresentar o pior resultado. Explicitar para o compilador como devem ocorrer os fluxos de dados entre CPU e GPU não surte efeito positivo quando a matriz de dados é muito pequena, mas ao aumentar o tamanho da matriz, indicar ao compilador a maneira

Tabela 6: Resultado, em milisegundos, das rodadas de teste de sensibilidade da variação do uso de diretivas e incremento do tamanho dos dados de entrada. Em verde o menor tempo de cada rodada e em vermelho o maior.

	Matriz 64X64	Matriz 640X640	Matriz 6400X6400
Kernel Puro	1272	5297	424752
Parallel Puro	1544	23237	1998382
Kernel Fluxo de Dados	1831	6534	326113
Parallel Fluxo de Dados	1901	5563	315772
CPU	1012	99395	10266200

como essa transição deve fluir acaba por obter o melhor resultado quando incrementa-se o tamanho da matriz de dados. Nota-se que o uso do Kernel Puro apresenta bons resultados, isso ocorre por que ao permitir que o compilador tome as decisões de como seria a melhor forma de paralelizar determinado trecho de código, ele pode encontrar melhores maneiras de lidar com o fluxo de dados entre CPU-GPU. O uso incorreto das diretivas OpenACC são evidenciadas no teste Parallel Puro, onde em um primeiro momento, com a matriz de dados pequena, não é perceptível, mas com o incremento da entrada de dados, fica mais evidente, contudo, obtendo melhores resultados em relação a CPU à partir da série Matriz 640X640. Para uma entrada de dados muito grande, como em Matriz 6400X6400, definir a forma de paralelismo e como devem ocorrer a transição de dados entre a CPU e a GPU demonstra o melhor resultado, ou seja, o menor tempo.

Essa série de testes demonstra o quanto o uso das diretivas OpenACC são sensíveis quanto ao tamanho da entrada de dados e a correta aplicação, indicando a necessidade de uma avaliação minuciosa no momento de inseri-las no contexto do modelo.

### Um Ensaio de Integração

Para melhorar o aspecto de tempo gasto na transferência de dados entre o processador e a placa gráfica, deve-se aproveitar o momento para realizar o maior número possível de operações sob o mesmo dado. Pensando nisso, o próximo passo foi implementar testes que se encaixassem nesse contexto. O trecho de código a seguir foi retirado da microfísica WSM5.

```
do k = kts, kte
  do i = its, ite
    if(t(i,k).lt.ttp) then
      xal(i) = xai
      xbl(i) = xbi
    else
      xal(i) = xa
      xbl(i) = xb
    endif
  enddo
do i = its, ite
```

```

tr=ttp/t(i,k)
logtr=log(tr)
qs(i,k,1)=psat*exp(logtr*(xa)+xb*(1.-tr))
qs(i,k,1) = ep2 * qs(i,k,1) / (p(i,k) - qs(i,k,1))
qs(i,k,1) = max(qs(i,k,1),qmin)
rh(i,k,1) = max(q(i,k) / qs(i,k,1),qmin)
qs(i,k,2) = psat*exp(logtr*(xal(i))+xbl(i)*(1.-tr))
qs(i,k,2) = ep2 * qs(i,k,2) / (p(i,k) - qs(i,k,2))
qs(i,k,2) = max(qs(i,k,2),qmin)
rh(i,k,2) = max(q(i,k) / qs(i,k,2),qmin)
enddo
enddo

```

Ao aplicar as técnicas para minimizar a transferência de dados entre o *host* e o *device*, o código torna-se algo como:

```

!$acc data copyout(qs(its:ite,kts:kte,1:2), rh(its:ite,kts:kte,1:2))
copyin(psat, ep2, ttp, t(its:ite, kts:kte), p(its:ite, kts:kte), xa,
xb, xai, xbi, qmin, q(its:ite, kts:kte))
!$acc parallel
  do k = kts, kte
    do i = its, ite
      if(t(i,k).lt.ttp) then
        qs(i,k,1) = max( (ep2 *
(psat*exp((log(ttp/t(i,k)))*(xa)+xb*(1.-(ttp/t(i,k)))))) / (p(i,k) -
(psat*exp((log(ttp/t(i,k)))*(xa)+xb*(1.-(ttp/t(i,k)))))) ), qmin)
        rh(i,k,1) = max(q(i,k) / qs(i,k,1),qmin)
        qs(i,k,2) = max(( ep2 *
(psat*exp((log(ttp/t(i,k)))*(xai)+xbi*(1.-(ttp/t(i,k)))))) / (p(i,k)
- (psat*exp((log(ttp/t(i,k)))*(xai)+xbi*(1.-(ttp/t(i,k))))))), qmin)
        rh(i,k,2) = max(q(i,k) / qs(i,k,2),qmin)
      else
        qs(i,k,1) = max( (ep2 *
(psat*exp((log(ttp/t(i,k)))*(xa)+xb*(1.-(ttp/t(i,k)))))) / (p(i,k) -
(psat*exp((log(ttp/t(i,k)))*(xa)+xb*(1.-(ttp/t(i,k)))))) ), qmin)
        rh(i,k,1) = max(q(i,k) / qs(i,k,1),qmin)
        qs(i,k,2) = max(( ep2 *
(psat*exp((log(ttp/t(i,k)))*(xa)+xb*(1.-(ttp/t(i,k)))))) / (p(i,k) -
(psat*exp((log(ttp/t(i,k)))*(xa)+xb*(1.-(ttp/t(i,k))))))), qmin)
        rh(i,k,2) = max(q(i,k) / qs(i,k,2),qmin)
      endif
    enddo
  enddo
!$acc end parallel
!$acc end data

```

Na *flag* “!\$acc data”, é indicada à API o comportamento que o programa deve adotar, em “copyout”, cria-se a referência de todas as variáveis que devem ser criadas diretamente na GPU e após a conclusão da computação, enviadas para a CPU, já em “copyin”, são definidas todas as variáveis que participam da computação e devem ser enviadas à GPU, não tornando-se necessário o retorno desses dados para a CPU, fazendo com que o tempo necessário para a transferência de dados seja reduzido. Percebe-se também que as variáveis “tr” e “logtr” são substituídas respectivamente por “ttp/t(i,k)” e  $\log(\text{ttp}/t(i,k))$ , excluindo assim variáveis temporárias, não tornando necessária a transferência de dados entre GPU e CPU. A aplicação desta técnica surtiu efeitos de até 57 vezes no ambiente de testes para grandes laços de repetição, ou seja, grandes quantidades de dados, conforme mostra a tabela 7. Onde em azul e laranja demonstram o tempo necessário para executar o trecho de código na GPU,

Tabela 7: Tomadas de tempo de processamento (ms), comparando CPU, GPU e GPU com uso de diretivas de controle de fluxo de dados, com múltiplas operações sob o mesmo dado. a) Matriz de dados de 200x200x27 elementos; b) Matriz de dados de 3200x3200x3 elementos.

a)

	t1	t2	t3
GPU Sem diretiva data	22083	20464	20502
GPU	21436	19656	19745
CPU	1129559	1134982	1134991

b)

	t1	t2	t3
GPU Sem diretiva data	838522	796065	796117
GPU	818430	774105	774208
CPU	32335308	32335261	32335356

sem e com o uso da diretiva *data* e em cinza, o tempo necessário para executar a computação na CPU.

Ao substituir esse trecho de código no algoritmo da microfísica, o tempo final de processamento do algoritmo foi maior. Na microfísica WSM5, os valores de k e i são

respectivamente 27 e 200, sendo assim, a matriz de dados é muito pequena. O tempo necessário para realizar todas as transferências de dados entre o *host* e o *device*, mais o tempo necessário para efetuar a computação, acaba sendo maior que o tempo gasto para efetuar toda a computação diretamente na CPU.

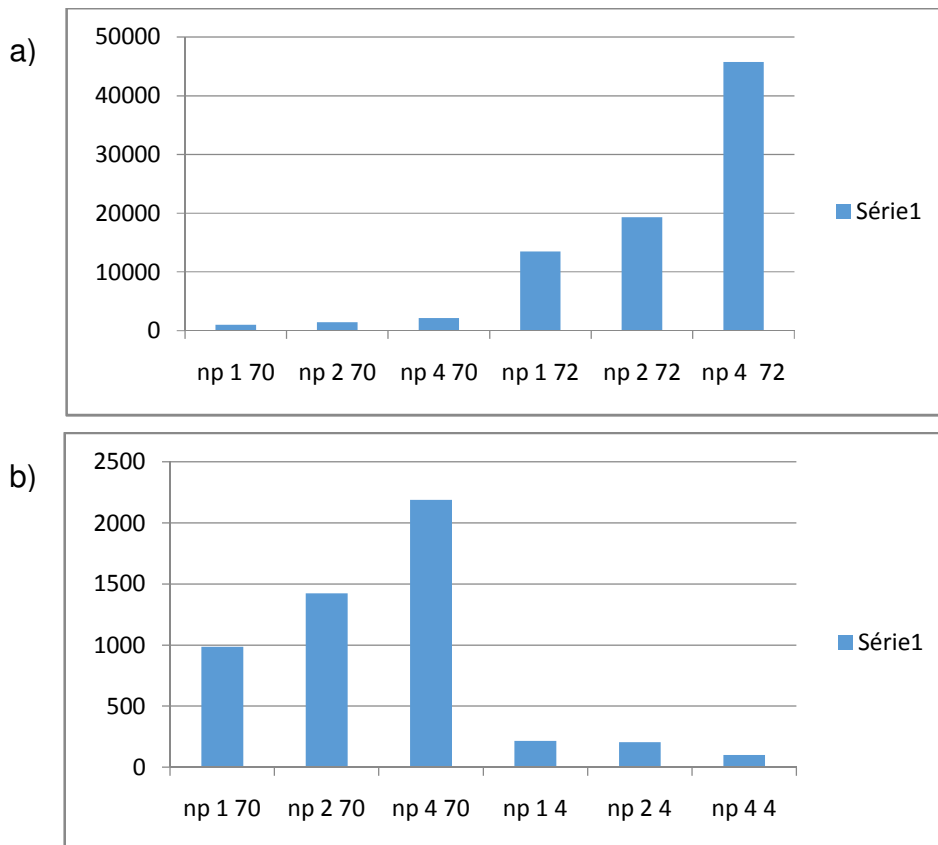


Figura 18: Tomada de tempo com adição nós, np significa o número de processadores a ser solicitado para a computação seguido do identificador da microfísica. a) Uso de GPU no processamento. b) Uso de GPU versus uso de CPU.

O modelo roda nativamente em MPI, permitindo assim a existência de possíveis múltiplos nós em processamento, dividindo os vetores de dados de maneira variável, não garantindo assim que o tamanho seja grande o suficiente para valer a pena processar na GPU e arcar com os tempos de transferência de dados. Outro fator relevante seriam as possíveis disputas de recurso, no caso, da placa gráfica. Ao atingir o trecho de código que exige o uso da GPU, diferentes processadores requisitam este recurso. Se este recurso encontra-se ocupado, processando o pedido de outro



processador, por exemplo, o processador entra em estado de espera até que o recurso esteja novamente disponível. Esses pequenos atrasos gerados da espera de recurso contribuem para a defasagem temporal da computação. Isso pode ser observado através de testes que incrementavam o uso de nós na execução do modelo, como mostra a figura 18.

Com o aumento de nós de processamento os trechos das microfísicas onde a placa gráfica é requisitada aumenta figura 18a, enquanto no caso do processamento somente utilizando CPU, o tempo diminui.

## CAPÍTULO 5: CONCLUSÃO

Foram averiguados o uso de tecnologias e técnicas de processamento para a realização de computação científica na análise de fenômenos atmosféricos. Ao estudar algoritmos com base em arquitetura SIMT, foram criadas propostas de algoritmos que fizeram uso de técnicas de paralelismo baseadas em GPU usando avaliação das parcelas de código que possuem características dessa arquitetura. Ao realizar a aplicação de *tags* da API OpenACC foram produzidos tempos de processamento maiores que aqueles gerados sem uso da técnica, isso demonstrou a dificuldade gerada pela tentativa de paralelizar código que foi criado especificamente para ser executado em uma arquitetura MIND. A grande sensibilidade relacionada ao tamanho da entrada de dados e o uso correto das diretivas também foi um fator determinante, uma vez recorre à necessidade de avaliar previamente e através de testes, quais as possíveis combinações que geram um melhor resultado temporal. Apesar das diretivas OpenACC serem um facilitador na aplicação de paralelismo, torna-se necessária uma avaliação minuciosa do tamanho dos dados e operações a serem executadas na GPU, uma vez que o tempo para transferências de dados entre o *host* e o *device* consome uma parcela de tempo de processamento a ser considerada quanto ao uso em pequenas operações. Por fim, esse tempo pode ser maior que a própria computação, sendo assim, melhor executar localmente na CPU.

Para viabilizar o uso de OpenACC para paralelizar o algoritmo da microfísica de nuvem WSM5, tornam-se necessárias expressivas alterações na maneira em que o algoritmo está escrito, através de uma avaliação que garanta a exclusão de variáveis temporárias, aumentando o tamanho dos dados e execuções sobre esses dados.

O modelo escolhido para o estudo possui todas as características necessárias para sustentar a pesquisa de aplicação de computação de alto desempenho baseada em GPUs, uma vez que possui características SIMT e descreve bem os aspectos de tempo da região. Foram criadas versões de algoritmos da microfísica de nuvens WSM5 com o uso de OpenACC, contudo, nenhum deles executou em um tempo menor que o algoritmo original, principalmente quando solicitado à executar sobre o domínio MPI,

uma vez que as incertezas geradas da proposta de um híbrido não foram totalmente avaliadas, acredita-se que, como citado no capítulo anterior, as condições de disputas obrigavam aos outros nós (*host*) a aguardar a liberação de recursos (*devices*).

Pretende-se, para dar continuidade às pesquisas realizadas, manter o foco no algoritmo de microfísica WSM5 utilizando a API OpenACC. Serão verificados os ganhos que podem ser alcançados quando o código da microfísica sofre grandes alterações para torna-se mais apropriado ao uso das diretivas. Assim que constatados os primeiros indícios de ganho em performance, será montada uma estrutura que viabilize o conceito de código híbrido MPI + CUDA, detalhando possíveis arranjos estruturais que minimizem as condições de disputa entre recursos, *host-devices*. Ao verificar a viabilidade do uso de uma estrutura tal como OpenACC como facilitador do uso de processamento com um possível alto grau de paralelismo, pretende-se propor uma ferramenta semelhante que funcione como uma versão aberta compatível com outros compiladores.

## BIBLIOGRAFIA

- AMDAHL, G. M., *The Logical Design of an Intermediate Speed Digital Computer*, unpublished PhD dissertation, Univ. of Wisconsin, Madison, Wisconsin, Estados Unidos, 1952.
- BERNSTEIN, A. J., *Program Analysis for Parallel Processing*, IEEE Trans. em Electronic Computers, EC-15, pp. 757–763, outubro de 1966.
- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. e STEIN, C. *Introduction to Algorithms*. Ed. 3, MIT Press, 2009.
- CULLER, D. E.; SINGH, J. P. e GUPTA, A., *Parallel Computer Architecture - A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.
- FLYNN, M., *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput. C-21: 948, 1972.
- FLYNN, L. J. *Intel halts development of 2 new microprocessors*. The New York Times, disponível em: <http://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007>, publicado em 8 de maio de 2004, último acesso em 20 de agosto de 2013.
- GAMBETÁ, M. R. S. L., *Circulações locais no Rio Grande do Sul: Brisas marítimas/terrestre e sua interação com as Brisas lacustres/terrestres*. Dissertação apresentada ao Programa de Pós Graduação em Modelagem Computacional da Universidade Federal do Rio Grande, Rio Grande, RS, 2013
- GOMES, C. S., GAMBETÁ, M., SPOLAVORI, A., KRUSCHE, N., *Estudo sobre a acurácia da previsão numérica gerada pelo WRF para a cidade de Rio Grande*, Anais do XVII Congresso Brasileiro de Meteorologia, Gramado, RS, 2012.
- GOMES, C. S., GAMBETÁ, M., SPOLAVORI, A., KRUSCHE, N., ROCHA, R., *Simulação de nevoeiros de advecção em Rio Grande, RS, Brasil: Análise das condições iniciais e de fronteira em modelo de mesoescala*, XI Congresso Argentino de Meteorologia, Mendoza, Argentina, 2012.
- HIBBARD, B. e PAUL, B., *CAVE5D Release 2.0*, University of Wisconsin – Madison, Wisconsin, Estados Unidos, disponível em: <http://www.mcs.anl.gov/~mickelso/CAVE2.0.html>. Acesso em: 24 de agosto 2012.

- HUANG, Y. e LI, G., *Descriptive models for internet of things*, International Conference on Intelligent Control and Information Processing (ICICIP), 2010, 483–486.
- HWU, W. W., KEUTZER, K. e MARTTSON, T., *The Concurrency Challenge*, IEEE Design and Test of Computers, 2008, 312-320.
- JOHNSON, C.R. e SANDERSON, A.R., “A New Step: Visualizing Errs and Uncertainty,” IEEE Computer Graphics and Applications, 23(5), 2003, 6-10.
- KOSHIZUKA, N. e SAKAMURA, K. *Ubiquitous id: Standards for ubiquitous computing and the internet of things*, IEEE Pervasive Computing, 9(4), 2010, 98 –101.
- KNUTH, D., *Big Omicron and Big Omega and Big Theta*, SIGACT News, abril-junho de 1976, p. 18-24.
- KRANZ, M., MÖLLER, A. e ROALTER, L., *Robots, objects, humans: Towards seamless interaction in intelligent environments*, in 1st International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2011), Algarve, Portugal, 2011.
- LIFTON, J., *Dual Reality: An Emerging Medium*, Ph.D. Dissertation, Massachusetts Institute of Technology, Department of Media Arts and Sciences, setembro de 2007.
- LIFTON, J. e PARADISO, J. A., *Dual Reality: Merging the Real and Virtual*, Proceedings of the First International ICST Conference on Facets of Virtual Environments (FaVE), julho de 2009.
- MCCORMICK, B.H., *Visualization in Scientific Computing*, T.A. DeFanti, and M.D. Brown, eds., ACM Press, 1987.
- NICHALAKES, J. e DUDHIA, D., *The weather research and forecast model: Software architecture and performance*, Mesoscale and Microscale Meteorology Division, National Center for Atmospheric Research, Boulder, Colorado 80307 U.S.A. ANO
- NVIDIA. *Cuda Programming Guide*, Santa Clara, CA: NVIDIA Corp., 2007.
- NVIDIA®, CUDA Parallel Computing Platform, Disponível em: [http://www.nvidia.com.br/object/cuda\\_home\\_new\\_br.html](http://www.nvidia.com.br/object/cuda_home_new_br.html). Último acesso em: 21 de agosto 2013.
- NVIDIA CUDA™: NVIDIA CUDA C Programming Guide. Version: 4.2, 2012.
- OpenMP, The OpenMP® API specification for parallel programming. Disponível em: <http://openmp.org/wp/> Último acesso em: 21 de agosto 2013.

Open MPI: Open Source High Performance Computing. Disponível em: <http://www.open-mpi.org/>. Último acesso em: 21 de agosto 2013.

PARADISO, J. A. e LANDAY, J. A., *Guest editors' introduction: Cross-reality environments*, IEEE Pervasive Computing, 8, 2009, 14–15.

PATT, Y., *The Microprocessor Ten Years From Now: What Are The Challenges, How Do We Meet Them? (wmv)*. Palestra na Universidade Carnegie Mellon, abril de 2004.

PATTERSON, D. A. e Hennessy, L. J., Ed. 2, *Organização e Projetos de Computadores - A interface hardware/software.*, Rio de Janeiro: LTC, 2005. 552 p.

SHEN, J. P. e LIPASTI, M. H., *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Professional, 2005, p. 561.

SUTTER, H. e LARUS, J. *Software and the concurrency resolution*. ACM Queue, 2005.

*The OpenACC™ Application Programming Interface*, Version 2.0, junho de 2013.

UID Center, "<http://www.uidcenter.org/>," Acessado em: abril de 2011.

VAN DAM, A., FORSBERG, A. ; LAIDLAW, D.H. ; LAVIOLA, J.J. ; Simpson, R.M., *Immersive VR for Scientific Visualization: A Progress Report*, IEEE Computer Graphics and Applications, 20(6), 2000, 26-52.

VAN DAM, A., LAIDLAW, D.H. e SIMPSON, R.M. *Experiments in Immersive Virtual Reality for Scientific Visualization*, Computers and Graphics, 26(4), 2002, 535-555.

VON NEUMANN, J., *First Draft of a Report on the EDVAC*. Contract No. W-670-ORD-4926, U.S. Army Ordnance Department and University of Pennsylvania, 1945.

WILKS, D.S., *Statistical Methods in the Atmospheric Sciences: an Introduction*. Academic Press, 1995, 467 pp.

MIELIKAINEN, J., Huang, B. Huang, H., Goldberg, M. Improved GPU/CUDA Based Parallel Weather and Research Forecast(WRF) Single Moment 5-Class(WSM5) Cloud Microphysics, IEEE Journal of Selected Topics in a Applied Earth Observations and Remote Sensing, vol. 5, n 4, agosto 2012.

WOLFE, M. OpenACC Kernels and Parallel Constructs. Disponível em: <http://www.pgroup.com/lit/articles/insider/v4n2a1.htm>. Último acesso em 18 de setembro 2013.

WOLFE, M. The PGI Accelerator Programming Model on NVIDIA GPUs Part 1, 2009.  
Disponível em: <http://www.pgroup.com/lit/articles/insider/v1n1a1.htm>. Último acesso em  
01 de outubro 2013.