MODELO COMPUTACIONAL PARALELO BASEADO EM GPU PARA CÁLCULO DO CAMPO DE VENTO DE UM SISTEMA DE DISPERSÃO ATMOSFÉRICA DE RADIONUCLÍDEOS

André Luís da Silva Pinheiro

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia Nuclear, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia Nuclear.

Orientador: Roberto Schirru

Rio de Janeiro
Março de 2017

MODELO COMPUTACIONAL PARALELO BASEADO EM GPU PARA CÁLCULO DO CAMPO DE VENTO DE UM SISTEMA DE DISPERSÃO ATMOSFÉRICA DE RADIONUCLÍDEOS

André Luís da Silva Pinheiro

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA NUCLEAR.

Examinada por:

_____

Prof. Roberto Schirru, D.Sc.

_____

Prof. Cláudio Márcio do Nascimento Abreu Pereira, D.Sc.

_____

Prof. Eduardo Gomes Dutra do Carmo, D.Sc.

_____

Prof. Ademir Xavier da Silva, D.Sc.

_____

Prof. Celso Marcelo Franklin Lapa, D.Sc.

_____

Prof. Cesar Marques Salgado, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2017

Pinheiro, André Luís da Silva

Modelo Computacional Paralelo baseado em GPU para Cálculo do Campo de Vento de um Sistema de Dispersão Atmosférica de Radionuclídeos/André Luís da Silva Pinheiro. – Rio de Janeiro: UFRJ/COPPE, 2017.

XXI, 193 p.: il.; 29,7 cm.

Orientador:Roberto Schirru

Tese (doutorado) – UFRJ/ COPPE/ Programa de Engenharia Nuclear, 2017.

Referências Bibliográficas: p. 184-193.

1. Campo de Vento. 2. GPU. 3. Dispersão Atmosférica de Radionuclídeos. I. Schirru, Roberto. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Nuclear. III. Título.

DEDICATÓRIA

Talvez não existam palavras suficientes e significativas que me permitam agradecer...

Aos meus pais Alfredo Guilherme de Barcellos Pinheiro e Alice da Silva Pinheiro por tudo que fizeram, pela ajuda incondicional e por todos os momentos de privação para que eu pudesse chegar aqui.

À minha esposa Tatiana Santos da Silva pela paciência e compreensão especialmente nos momentos em que as obrigações se sobrepunham ao lazer.

A vocês, apenas posso expressar através da limitação destas meras palavras, e com elas lhes prestar esta humilde, mas sincera, homenagem.

Meu muito obrigado.

AGRADECIMENTOS

Ao professor Roberto Schirru pela oportunidade concedida e por sua orientação e amizade ao logo desta jornada.

Ao professor Cláudio Márcio do Nascimento Abreu Pereira, por sua amizade, pelo incentivo de iniciar o Doutorado e por sua grande paciência na orientação ao longo destes anos.

Aos meus pais, Alfredo e Alice, os quais amo muito e agradeço todos os dias por tê-los como pais.

A minha querida esposa que ao longo dos anos sempre me apoiou em todas minhas decisões.

Aos amigos que fiz no Laboratório de Monitoramento e Processo (LMP) no PEN/COPPE/UFRJ.

E a todos os demais que direta ou indiretamente me ajudaram a realizar este trabalho.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

MODELO COMPUTACIONAL PARALELO BASEADO EM GPU PARA CÁLCULO DO CAMPO DE VENTO DE UM SISTEMA DE DISPERSÃO ATMOSFÉRICA DE RADIONUCLÍDEOS

André Luís da Silva Pinheiro

Março/2017

Orientador: Roberto Schirru

Programa: Engenharia Nuclear

Com o objetivo de melhorar a previsão da dispersão atmosférica de radionuclídeos (DAR) nas proximidades da Central Nuclear Brasileira Almirante Álvaro Alberto (CNAAA), está em desenvolvimento um sistema computacional mais refinado. Para alcançar o refinamento desejado, o esforço computacional necessário aumenta de tal forma que a execução do sistema pelos computadores atuais leva a um tempo de processamento proibitivo. Com o objetivo de acelerar a execução de tal sistema refinado, permitindo seu uso efetivo na previsão em tempo real da DAR, uma abordagem paralela baseada em GPU foi proposta. Basicamente, o sistema DAR usado no CNAAA é composto por 4 módulos principais (programas): Módulo Termo Fonte, Campo de Vento, Dispersão de Pluma e Módulos de Projeção de Pluma.

Este trabalho é focado no módulo do Campo de Vento, que utiliza uma abordagem não divergente, com base no modelo *Winds Extrapolated from Stability and Terrain* (WEST). Devido à forte natureza sequencial do algoritmo, a decomposição do domínio por um particionamento 3D-Red-Black foi proposto e um novo algoritmo baseado em GPU paralelo foi implementado usando o Compute Unified Device Architecture (CUDA) e a linguagem de programação C. Como resultado, o tempo de execução de uma simulação de malha fina diminuiu de cerca de 450 segundos (executado em um Intel-I7) a 5,60 segundos (em execução em uma GPU GTX-680). Aqui, são apresentadas e discutidas as questões mais importantes da implementação paralela, sua otimização, bem como os resultados comparativos.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

GPU-BASED COMPUTING PARALLEL MODEL FOR CALCULATING THE WIND FIELD OF A RADIONUCLIDE ATMOSPHERIC DISPERSION SYSTEM

André Luís da Silva Pinheiro

March/2017

Advisor: Roberto Schirru

Department: Nuclear Engineering

In order to improve the prediction of atmospheric dispersion of radionuclide (ADR) in vicinity of Central Nuclear Almirante Alvaro Alberto (CNAAA) Brazilian Nuclear Power Plants (NPP), a more refined computational system is under development. To achieve desired refinement, the required computational effort increases in such a way that system's execution by current computers leads to prohibitive processing time. Aiming to accelerate execution of such refined system, allowing its effective use in real-time prediction of ADR, a GPU-based parallel approach has been proposed. Basically, the ADR system used in CNAAA is comprised by 4 main modules (programs): Source Term, Wind Field, Plume Dispersion and Plume Projection modules.

This work is focused on the Wind Field module, which uses a mass-consistent approach, based on *Winds Extrapolated from Stability and Terrain* (WEST) model. Due to the strong sequential nature of the algorithm, domain decomposition by a 3D-Red-Black partitioning was proposed and a new parallel GPU-based algorithm was implemented using the Compute Unified Device Architecture (CUDA) and C programming language. As a result, the execution time of a fine-grained simulation decreased from about 450 seconds (running on an Intel-I7) to 5,60 seconds (running on a GTX-680 GPU). Here, the most important issues of the parallel implementation, their optimization, as well as comparative results, are presented and discussed.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS, ACRÔNIMOS E DEFINIÇÕES

AIEA – Agência Internacional de Energia Atômica

CFD – Computational Fluid Dynamics

CLA – Camada Limite Atmosférica

CNAAA – Central Nuclear Almirante Álvaro Alberto

CPU – Unidade de Processamento Central

CUDA – Compute Unified Device Architecture

DAR – Dispersão Atmosférica de Radionuclídeos

EPA – United States Environmental Protection Agency

FORTRAN – IBM Mathematical FORmula TRANslation System

GPU – Graphics Processor Unit (Unidade Gráfica de Processamento)

MPI – Message Passing Interface

PSO – Particle Swarm Optimization

PVM – Parallel Virtual Machine

SCA – Sistema de Controle Ambiental

SCRAM – Support Center for Regulatory Atmospheric Modeling

UDM – Urban Dispersion Model

ULA – Unidade Lógico Aritmética

CAPÍTULO 1. INTRODUÇÃO

1.1. APRESENTAÇÃO DO PROBLEMA

De acordo com a trigésima terceira edição do relatório *Energy, Electricity and Nuclear Power estimates for the Period up to 2050*, publicado pela IAEA (INTERNATIONAL ATOMIC ENERGY AGENCY, 2013, p. 21), os reatores nucleares em 2012 foram responsáveis por 11,3% da produção de energia elétrica no mundo e, em 2020 estima-se que esta produção possa chegar em até 13,9%. Já em 2012, a energia nuclear foi a quarta maior fonte de energia; ficando atrás do carvão, dos combustíveis líquidos e do gás natural. Apesar de ser uma energia limpa[1], o perigo se encontra no material de alta radioatividade necessário ao funcionamento das usinas e na possibilidade de acidente, que podem ser devastadores.

Apesar de todos os esforços e tecnologias empregadas na segurança das usinas nucleares, diversos acidentes ocorreram ao longo dos anos. O acidente ocorrido na Ucrânia, na usina nuclear de Chernobyl em 26 de abril de 1986 é considerado o maior acidente nuclear da história da humanidade. O acidente recebeu a classificação máxima de gravidade, o de nível 7, considerado o mais grave da Escala Internacional de Acidentes Nucleares (INES). O acidente aconteceu em razão do rompimento do reator de uma usina durante um superaquecimento, o que causou duas explosões e um incêndio de grandes

[1] Quando se fala em "energia limpa", não está se falando de um tipo de geração de energia que não cause nenhum impacto ambiental. Energia limpa refere-se àquela fonte de energia que não lança poluentes na atmosfera, interferindo no ciclo do carbono, e que apresenta um impacto sobre a natureza somente no local da instalação da usina.

proporções liberando na atmosfera toneladas de radionuclídeos. A pluma radioativa foi arrastada pelo vento e se espalhou pela União Soviética, Europa Oriental, Escandinávia e Reino Unido, contaminando quase três quartos da Europa e matando milhares de pessoas.

O acidente nuclear na usina de Three Mile Island perto de Harrisburg, capital da Pensilvânia, Estados Unidos, ocorreu no dia 29 de setembro de 1979 e é considerado o maior acidente nuclear da história dos Estados Unidos, atingindo o nível 5 na Escala Internacional de Acidentes Nucleares (INES). Após a quebra da bomba de água de um reator, o núcleo do reator sofreu uma fusão parcial. Para evitar uma explosão, os técnicos tiveram que liberar vapores e gases radioativos na atmosfera contaminando assim o ambiente.

Já na Rússia em 6 de abril de 1993 uma explosão na usina de reprocessamento de combustível irradiado em Tomsk-7, em uma cidade da Sibéria Ocidental, hoje chamada de Seversk, provocou a formação de uma nuvem com a projeção de materiais radioativos. O número de vítimas é desconhecido e a cidade é fechada e só pode ser visitada a convite do governo.

Após ter parte de seu território devastado por um terremoto de 8,9 graus na escala Richter seguido de um tsunami no dia 11 de março de 2011, os japoneses enfrentaram um vazamento de radiação na usina nuclear de Fukushima. Após o derretimento de três dos seis reatores nucleares, a usina começou a liberar grandes quantidades de materiais radioativos, tornando-se o maior desastre nuclear desde o acidente nuclear de Chernobil atingindo o nível 7 da Escala Internacional de Acidentes Nucleares (INES).

É difícil não ficar perplexo diante de tais consequências que os acidentes nucleares deixam. São diversas as vítimas em um acidente nuclear, contando com mortos e doentes que tiveram contato direto ou indireto com o material radioativo.

Os acidentes com usinas nucleares geralmente envolvem a liberação de radiações que podem ser prejudiciais para as pessoas, meio ambiente e a própria central nuclear. A dispersão atmosférica e a deposição de radionuclídeos podem impor altas taxas de dose à população local, juntamente com a contaminação do ar, água, solo, plantas e animais, que também podem afetar a saúde humana por contato direto, inalação e ingestão.

As emissões dos acidentes das centrais nucleares apresentam muitos produtos de fissão prejudiciais e outros radionuclídeos de sua cadeia de decaimento, sendo ^{137}Cs e ^{131}I os mais investigados devido ao seu grande impacto na saúde humana e concentrações significativas. Outros radionuclídeos, tais como ^{133}Xe, ^{140}Ba, ^{140}La, ^{90}Sr, também são comuns em liberações de acidentes em centrais nucleares. A estimativa da concentração de ^{131}I é muito importante na fase inicial da liberação devido à sua meia-vida curta de apenas 8 dias. Por outro lado, os efeitos a longo prazo são causados pela contaminação por ^{137}Cs, que possui meia-vida de 30 anos e podem contaminar gerações futuras se os devidos cuidados não forem tomados.

Em acidentes severos, grandes quantidades de radionuclídeos são liberadas na forma de aerossol (^{137}Cs e ^{131}I, por exemplo) e gás (^{133}Xe, por exemplo), formando uma pluma radioativa, que é transportada pelo vento. De acordo com a característica de liberação (velocidade, temperatura, dimensões, altitude, altura, etc.) e as condições meteorológicas (velocidade do vento, direção, estabilidade, temperatura, índice de precipitação, etc.), a pluma pode ser transportada ao longo de muitos quilômetros, afetando as pessoas e o meio ambiente em uma grande área, por dias, meses ou anos

causando problemas para a população que em algumas vezes sentem de imediato as consequências, outras vezes, ficam com sequelas como por exemplo o câncer de tireoide.

As sequelas deixadas por esse tipo de acidente levam um tempo indeterminado para desaparecer totalmente, o que explica a dificuldade mundial em esquecer tais tragédias. O plano de evacuação da população no caso de um acidente com dispersão de radionuclídeos na atmosfera, deve levar em consideração a trajetória da nuvem radioativa, para que medidas de proteção a população nas proximidades do acidente sejam tomadas de forma a minimizar os danos causados pelo acidente.

Por isso, é muito importante avaliar o risco devido a acidentes em centrais nucleares, a fim de facilitar a preparação e aconselhar a equipe tomadora de decisões. Muitos autores, como (CHRISTOUDIAS, PROESTOS e LELIEVELD, 2014), (ARNOLD, GUFLER, *et al.*, 2012) e (LELIEVELD, KUNKEL e LAWRENCE, 2012) focalizam suas pesquisas na estimativa do risco devido às liberações e à dispersão atmosférica de radionuclídeos provenientes de acidentes com centrais nucleares.

Nesse contexto, os modelos de dispersão atmosférica são ferramentas importantes para prever o impacto de eventuais liberações radioativas das centrais nucleares. A previsão rápida e precisa é crucial para conduzir decisões relacionadas à proteção de pessoas e sua evacuação da área afetada.

A previsão da dispersão atmosférica de radionuclídeos (DAR) envolve a simulação de modelos físicos complexos que consomem tempo e, às vezes, para atingir o refinamento e a precisão desejados, o esforço computacional necessário aumenta de tal forma que a execução do sistema pelos computadores atuais leva a um tempo de processamento proibitivo.

O Sistema de Controle Ambiental (SCA), que é um sistema de previsão da dispersão atmosférica de radionuclídeos (DAR), utilizado na Central Nuclear Almirante Álvaro Alberto (CNAAA) em Angra dos Reis, litoral do Rio de Janeiro, foi desenvolvido nos anos 80, considerando várias simplificações e aproximações para permitir sua execução em tempo real nos computadores disponíveis na época. Algumas simplificações importantes foram:

i) Representação em baixa resolução espacial do domínio computacional;
ii) Critérios de parada dos métodos numéricos baseados no número de iterações;
iii) O modelo de difusão se utiliza de um modelo de bufadas[2].

Mesmo com tais simplificações sendo feitas, ainda era gasto uma grande quantidade de tempo para a execução da simulação. O Sistema de Controle Ambiental, é composto por 4 módulos principais (programas): Termo de Fonte, Campo de Vento, Dispersão de Pluma e Módulos de Projeção.

O módulo Termo Fonte é responsável pela previsão de concentrações e taxas de liberação de material nuclear com base no inventário atual e nos status da central nuclear (incluindo, se aplicado, o acidente diagnosticado). Um campo de vento não divergente é calculado pelo módulo Campo de Vento, que é um modelo de diagnóstico (HOMICZ,

---

[2] O modelo de bufadas (*puff model*) é um modelo usado para ajudar a prever como a poluição do ar se dispersa na atmosfera onde os agentes poluidores (no caso os radionuclídeos), são dispersos na fonte em forma de bufadas em intervalos de tempo e não de forma continua, ou seja, a liberação contínua de material pode ser representada pela emissão de uma sequência de bufadas (*puffs*) discretas.

2002) baseado no *Winds Extrapolated from Stability and Terrain* (WEST), um algoritmo da *California Air Resources Board*.

WEST é um submodelo do código DEPICT, que a formulação e o manual do usuário são vistos em (FABRICK, SKLAREW e WILSON, 1976). (FABRICK, SKLAREW e WILSON, 1977) e possuem uma descrição muito boa do modelo WEST, que tem sido utilizado no presente trabalho, já que foi a referência utilizada no desenvolvimento do sistema SCA original da CNAAA. Ainda no módulo do Campo de Vento, o domínio computacional é discretizado em uma grade tridimensional e, para cada célula (ou nó) da grade, os campos de vento sem divergência são calculados por interpolação e extrapolação de dados observados, seguido de um procedimento de eliminação de divergência.

O módulo Dispersão da Pluma calcula a distribuição média de radionuclídeos, usando um modelo de bufadas com difusão gaussiana e trajetória lagrangeana variável definida por um campo de vento livre de divergência tridimensional. E finalmente, o módulo de Projeção faz estimativas simplificadas de dispersões de plumas de até 2 horas à frente.

O Sistema de Controle Ambiental da CNAAA cobre uma área de aproximadamente (17 x 11) quilômetros na vizinhança da central nuclear, como mostrado

na Figura 1, em que a posição da central nuclear (CNAAA), bem como as 4 estações meteorológicas (A, B, C e D) podem ser vistas.

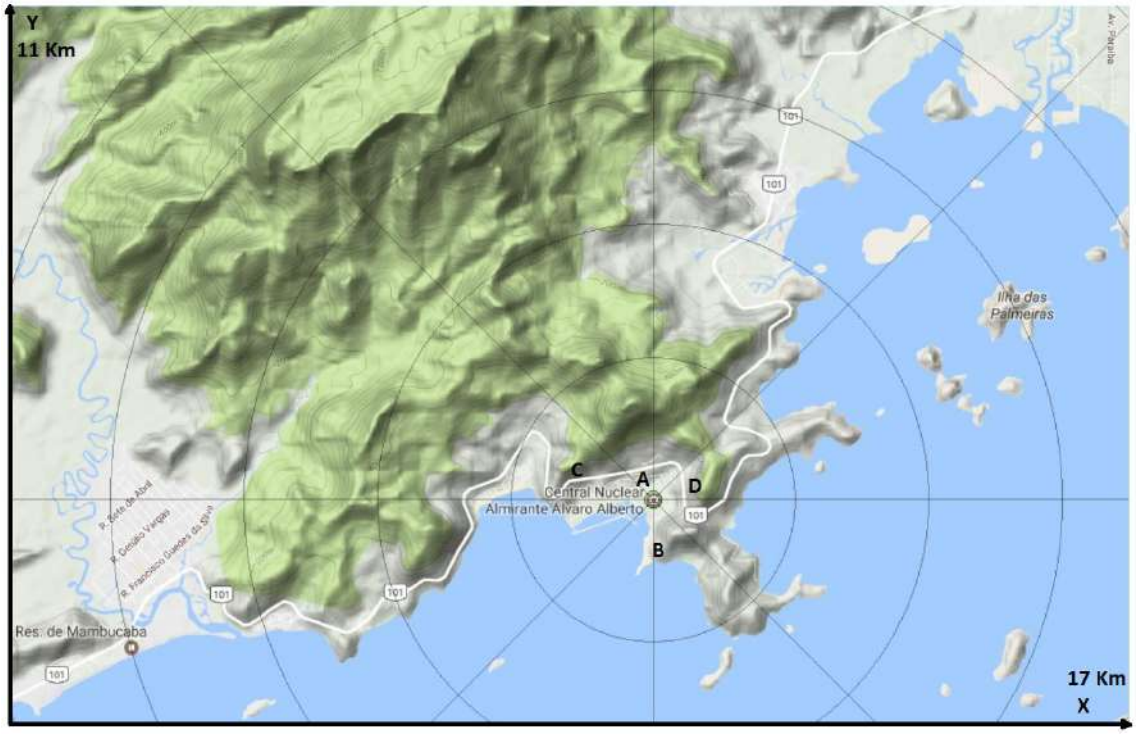

Figura 1 – Localização da CNAAA e região coberta pelo SCA

Atualmente, o domínio computacional é composto por 23.048 (67 x 43 x 8) células com áreas horizontais de 250 x 250 metros e dimensões verticais variáveis, cobrindo uma área total de aproximadamente17 x 11 quilômetros. Apesar do fato das dimensões não serem tão refinadas, outras aproximações relevantes são ainda observadas nos módulos de Campo de Vento e Dispersão de Plumas. Os modelos de transporte e difusão utilizados no módulo Dispersão de Plumas consideram a representação em baixa resolução espacial (por exemplo, considera 1 sopro por minuto e um máximo de cerca de 300 sopros), bem como outras simplificações numéricas.

Outra aproximação pode ser observada no algoritmo de minimização da divergência utilizado no módulo Campo de Vento, que utiliza um critério de convergência muito grosseiro (são consideradas 56 iterações fixas em vez de critérios de parada /

convergência baseados na divergência). E, finalmente, mesmo aplicando todas as simplificações acima mencionadas, era necessário um tempo-passo de 15 minutos para executar todo o sistema SCA.

1.2. DEFINIÇÃO DO PROBLEMA

Nas usinas nucleares brasileiras em operação, a determinação do campo de vento é feita, até então, considerando a média das velocidades medidas nas torres em um intervalo de tempo de 15 minutos e utilizando modelos de baixa resolução espacial, para reduzir o custo computacional envolvido, devido a capacidade de processamento dos computadores disponíveis na ocasião de seu desenvolvimento.

Devido a estas restrições computacionais exigidas na época, a resolução espacial do SCA é equivalente as resoluções utilizadas em modelos de dispersão atmosférica de longo alcance, como por exemplo o sistema ARGOS utilizado na Austrália (AUSTRALIAN GOVERNMENT, 2008) que quando necessita de um maior detalhamento (zoom), possui células com dimensões de 250 metros conforme ilustrado na Figura 2.

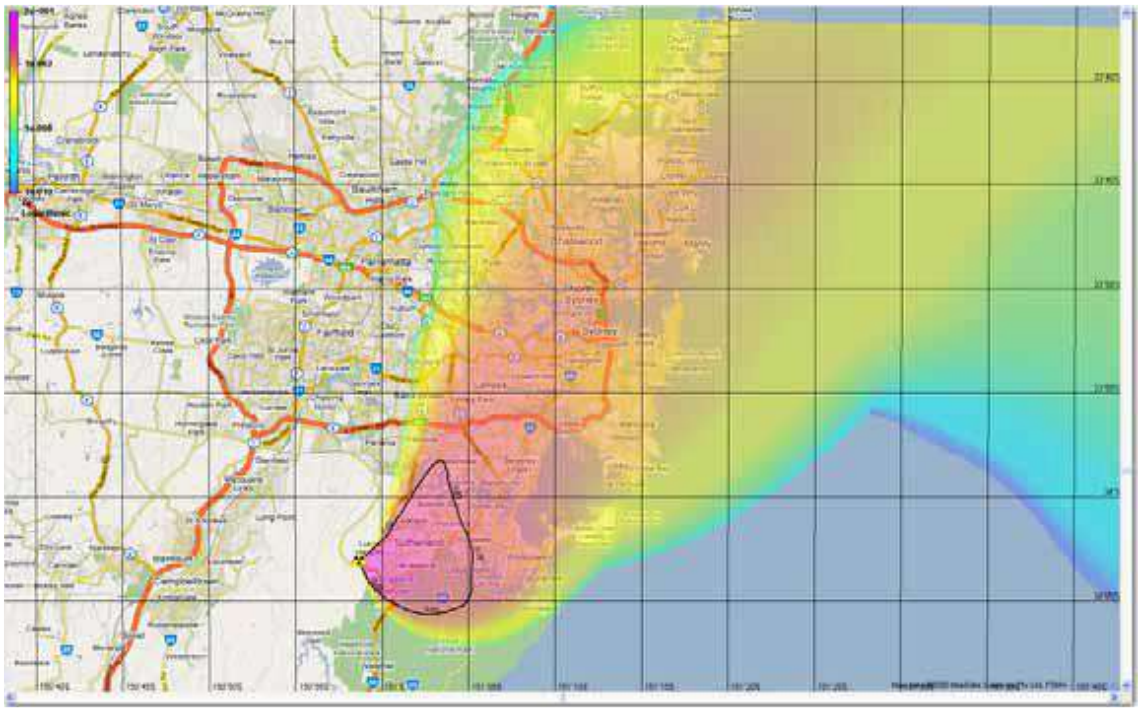

Figura 2 – Células com 250 metros utilizados no ARGOS
Fonte: Evaluation of ARGOS for use in Australia, ARPANSA Technical Report Nº 150, Pg121

Segundo (PEDERSEN, LEACH e HANSEN, 2007), regiões que possuem um número elevado de variações em sua topologia como é o caso de Angra dos Reis, devem preferencialmente utilizar o Modelo de Dispersão Urbana (UDM) (HALL, SPANTON, *et al.*) que foi desenvolvido para estimar a dispersão de bufadas (puffs) de contaminantes do ar a curtas distâncias em áreas urbanas. Para estas regiões, (PEDERSEN, LEACH e HANSEN, 2007) recomendam uma resolução espacial preferencialmente menor que 100 metros quadrados de forma que as pequenas elevações do terreno possam alterar os padrões de dispersão do campo de vento (Figura 3).

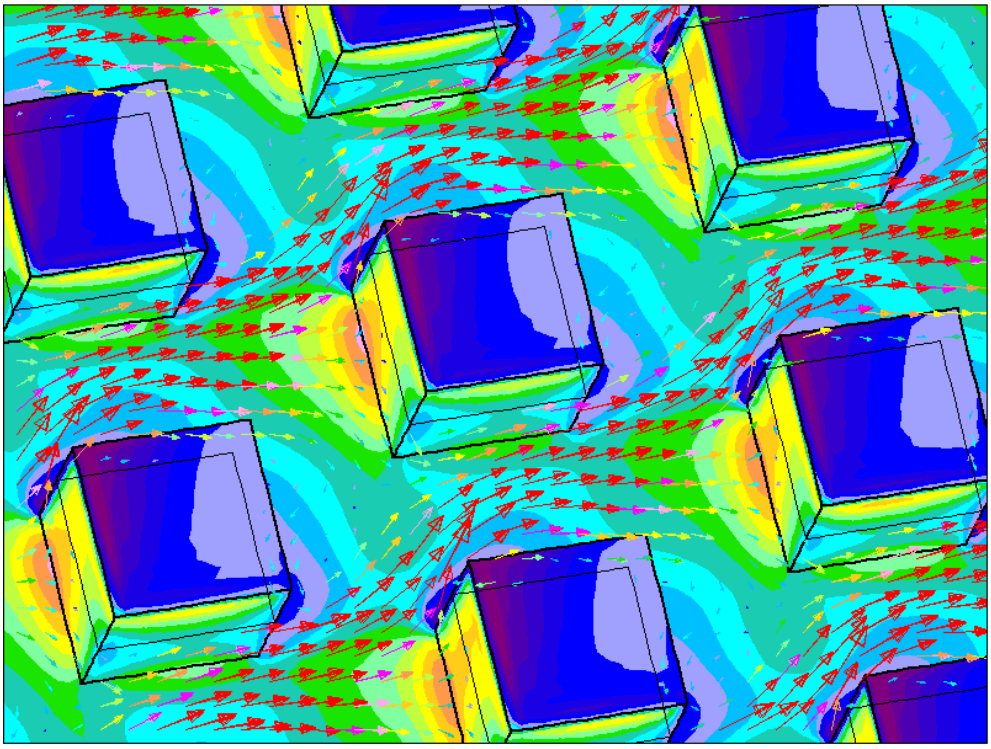

Figura 3 – Dispersão em regime urbano
Fonte: Adaptação de http://cmg.soton.ac.uk/research/projects/wind-direction-effects-on-urban-flows, acessado em 16/02/2017

Atualmente, o SCA possui uma resolução espacial de 62.500 m^2, que muito se afasta dos 100 m^2 recomendados. Desta forma, o ajuste da resolução espacial do SCA, para uma resolução compatível a uma boa tomada de decisão por partes das equipes de evacuação se faz necessário, entretanto, considerando a execução sequencial do sistema, com o ajuste da malha tridimensional, o custo computacional aumenta significativamente para valores inaceitáveis para tomadas de decisão em tempo real, mesmo nos computadores atuais.

Segundo (ALMEIDA, 2009), algoritmos paralelos baseados em GPU passam a ser uma solução viável em tais problemas de alto custo computacional. Entretanto, na concepção dos algoritmos paralelos, é necessário um estudo detalhado dos modelos matemáticos envolvidos, de forma a se definir as partes do programa que podem ser

executadas em paralelo. Tal tarefa, por sua vez é não trivial, quando os modelos possuem acoplamentos, como ocorre entre as células da malha a ser calculada.

1.3. OBJETIVOS

Visando melhorar o SCA da CNAAA, o desenvolvimento de uma abordagem computacional com uma melhor resolução espacial e precisa foi desenvolvida neste trabalho. O uso de computadores modernos já permite uma execução mais rápida do SCA como ele é, entretanto, para remover todas as simplificações anteriormente mencionadas, o custo computacional ainda é muito alto mesmo para os atuais computadores. Para superar tal limitação, foi utilizado o uso de técnicas de computação paralela. Após análises do algoritmo original, observou-se que poderiam ser feitas melhorias por meio da computação paralela que geraria grande impacto em tempo computacional, especialmente nos módulos Dispersão de Plumas, Projeção e Campo de Vento.

O presente trabalho foca no desenvolvimento de uma abordagem paralela otimizada para o módulo do Campo de Vento, que tem demonstrado um consumo excessivo de tempo se o refinamento desejado for considerado na simulação. O objetivo principal deste trabalho é reduzir o tempo de processamento dos cálculos do campo de vento utilizando uma abordagem paralela, mas com a preocupação de reproduzir os mesmos resultados (ou o mais próximo possível) obtidos pelo algoritmo original, que foi validado pela Eletronuclear para uso na CNAAA.

Investigações preliminares revelaram que para uma simulação realista refinada usando um domínio computacional composto por 1.475.072 (536 x 344 x 8) nós (células de cerca de 30m x 30m), o tempo de execução do módulo campo de vento (em um processador Intel Core I7) foi de cerca de 450 segundos. Considerando que um tempo de

execução estimado razoável para o módulo Campo de Vento funcionar corretamente em aplicações em tempo real deve ser de poucos segundos, o tempo de execução está longe de ser adequado para aplicações em tempo real.

Neste trabalho desenvolveu-se uma abordagem paralela baseada em GPU. Para isso, utilizou-se a decomposição do domínio por uma partição tridimensional Red-Black (FREEMAN e PHIILIPS, 1992); (YAVNEH, 1995), a resolução espacial do modelo já existente no SCA foi aumentada e utilizando a linguagem de programação CUDA (NVIDIA CORPORATION, 2014) (BUCK, 2007) um novo algoritmo paralelo foi implementado e otimizado por enxames de partículas (PSO) (KENNEDY e EBERHART, 1995). As questões mais importantes, os ganhos e limitações da implementação paralela, da otimização bem como os resultados comparativos, são apresentados e discutidos neste trabalho.

1.4. MOTIVAÇÃO

O presente trabalho foi motivado pela possibilidade de uma melhora na tomada de decisão, levando em consideração os padrões de dispersão do campo de vento em pequenas elevações no terreno que, até então, são desconsideradas pelo sistema atual.

Outro fator motivador é que com a utilização da computação paralela torna-se possível a utilização prática, em tempo real, de um modelo com resolução compatível ao de dispersão urbana, requeridos pelo SCA das usinas nucleares de Angra 1 e Angra 2.

1.5. TRABALHOS RELACIONADOS E CONTEXTUALIZAÇÃO

Os modelos tridimensionais de campos de vento, revisados por (HOMICZ, 2002), podem ser classificados em:

i) Modelos interpolados, os mais simples, nos quais os valores observados são apenas interpolados / extrapolados para todo o domínio computacional;

ii) Modelos diagnósticos (também conhecidos como modelos não divergentes) que tentam ajustar um campo de vento interpolado de modo que a equação de continuidade para o fluxo incompressível seja satisfeita em cada ponto da grade;

iii) Modelos linearizados, que tentam resolver, não apenas as equações de continuidade, mas também as equações de momentum de estado estacionário e;

iv) Modelos prognósticos, que são sofisticadas soluções numéricas, que podem consumir dezenas de horas ou dias.

Descartando os modelos interpolados devido à sua simplicidade e os modelos prognósticos devido à sua enorme demanda computacional, (HOMICZ, 2002) concluiu que os modelos diagnósticos são mais adequados, do que os linearizados, para prever a dispersão atmosférica de materiais perigosos em cenários de resposta de emergência.

Seguindo esta classificação, WEST, o modelo do campo de vento utilizado neste trabalho, é um modelo de diagnóstico ou não divergente, portanto, é apropriado para uso em sistemas de resposta de emergência. Além disso, modelos de diagnóstico têm sido utilizados em vários sistemas DAR, tais como em RASCAL 4.0 (NUCLEAR REGULATORY COMMISSION, 2012); (RENTAI, 2011), WindNinja (FORTHOFER, SHANNON e BUTLER, 2009) (FORTHOFER, SHANNON e BUTLER, 2010) e outros. (KOVALETSA, KOROLEVYCHC, *et al.,* 2013) relatou vantagens do uso de modelos diagnósticos também na dispersão atmosférica em microescala.

Embora sejam qualificados para a aplicação proposta, os modelos de campo de vento de diagnóstico podem tornar-se muito demorados dependendo da dimensão do domínio computacional. Para superar tal limitação, (SANJUAN, BRUN, *et al.*, 2014) propôs usar a partição de mapa, para permitir que a simulação seja dividida e executada por muitos computadores. Em 2015, eles enfrentaram novamente problemas com os tempos de execução e precisaram adaptar a resolução do mapa em um problema semelhante (SANJUAN, MARGALEF e CORTÉS, 2015) usando o programa de campo WindNinja (FORTHOFER, SHANNON e BUTLER, 2009).

Ainda hoje, o custo computacional relacionado ao cálculo do campo de vento é uma restrição. Em trabalhos muito recentes, (SANJUAN, MARGALEF e CORTÉS, 2016) ainda estão investigando novas abordagens para acelerar simulações de campo de vento. Neste artigo, o método Schur de decomposição de domínio (LAUB, 1979), (BARTH, CHAN e TANG, 1998) é proposto para acelerar cálculos de campos de vento e como resultado, eles alcançaram um *speedup* de cerca de 5,5 vezes executando o sistema em um cluster de computadores de 10 nós, diminuindo o tempo de simulação de 496s para 90s utilizando uma malha de 800 x 800 (160.000) células.

O processamento paralelo em clusters de computadores tem sido amplamente aplicado para acelerar simulações que consomem muito tempo em diversos campos. Algumas aplicações em engenharia nuclear (WAINTRAUB, SCHIRRU e PEREIRA, 2009), (PEREIRA e SACCO, 2008), incluindo a dispersão atmosférica de radionuclídeos (DE SAMPAIO, JUNIOR e LAPA, 2008) são relatadas. Deve-se enfatizar que, no trabalho mais recente, foi investigada uma abordagem de dinâmica de fluidos computacional (CFD) para dispersão em escala local de radionuclídeos, entretanto o cálculo do campo de vento não foi utilizado.

Uma tecnologia mais recente e mais barata na computação paralela é o uso de unidades de processamento gráfico (GPU) para computação de propósito geral. Tal abordagem também tem sido explorada em muitos campos. Alguns exemplos na área nuclear são a recarga de reatores (HEIMLICH, SILVA e MARTINEZ, 2016), o transporte de nêutrons (PEREIRA, MOL, *et al.*, 2013) e (HEIMLICH, MOL e PEREIRA, 2011). As implementações de GPU na modelagem da dispersão atmosférica também foram investigadas (SINGH, PARDYJAK, *et al.*, 2011), (HARVEY, HAMEED e VANDERBAUWHEDE, 2014), mas sem se concentrar no cálculo do campo de vento. Pode haver algumas outras investigações de sistemas de dispersão atmosférica acelerados por GPU, no entanto, neste trabalho, o interesse é restrito somente aos cálculos de campos de vento não divergentes.

Até a presente data, o autor não encontrou artigos de revistas enfocando diagnósticos utilizando simulações de campo de vento acelerados por GPU. No entanto, uma recente tese de bacharelado da *Universitat Autònoma de Barcelona* (Medina e Cortés, 2015) investigou o uso da GPU para acelerar o cálculo do campo de vento. O trabalho relata acelerações de cerca de 40 vezes usando um modelo de GPU GTX-TITAN para acelerar o código de campo de vento WindNinja. O referido documento é, contudo, demasiado curto e não fornece detalhes aprofundados sobre a modelação.

O modelo e / ou o método numérico utilizado pode ter grande influência sobre a facilidade ou a dificuldade de alcançar um paralelismo eficiente. Alguns algoritmos têm paralelização trivial devido à independência natural de seus processos, como os modelos de campo de vento interpolados. Outros, por outro lado, apresentam natureza sequencial, levando a soluções paralelas não diretas. Nestes casos, as técnicas de decomposição do

domínio podem ser utilizadas para modificar o algoritmo, de modo a facilitar a paralelização.

Entre os autores aqui citados, Sanjuan utilizou o método Schur de decomposição de domínio (LAUB, 1979), (BARTH, CHAN e TANG, 1998) aplicado a um diagnóstico de paralelização do campo de vento. (HEIMLICH, MOL e PEREIRA, 2011) usou um particionamento 2D-Red-Black aplicado a uma solução baseada em Gauss-Seidel de um problema de transferência de calor. No presente trabalho, propõe-se uma abordagem 3D-Red-Black para o algoritmo de minimização da divergência.

1.6. ORIGINALIDADE

O desenvolvimento de um modelo computacional paralelo, com resolução compatível ao da dispersão urbana, utilizando GPU para o cálculo do campo de vento tridimensional não divergente, para utilização em um sistema de dispersão atmosférica de radionuclídeos. O modelo 3D-Red-Black e sua utilização como abordagem para a partição do domínio computacional no algoritmo de minimização da divergência. A otimização da alocação de threads, utilizando-se do algoritmo de otimização por enxame de partículas, possuem em si próprios um caráter inovador e até onde vai nosso conhecimento não existe outra pesquisa nestas áreas.

1.7. JUSTIFICATIVA E RELEVÂNCIA

Na engenharia nuclear, diversos problemas apresentam soluções que demandam simulações computacionais de alto custo. A exemplo disso, pode-se citar os clássicos problemas de projeto neutrônico e termo hidráulico (DANIELA MAIOLINO, 2011) e a otimização da recarga de combustível nuclear (NICOLAU, 2014), (OLIVEIRA, 2013).

Seguindo a mesma linha de simulações computacionais de alto custo e as restrições impostas pelos sistemas computacionais da época, no final da década de 80, o Sistema de Controle Ambiental (SCA) foi desenvolvido e para que o sistema fosse computacionalmente viável, uma série de restrições se fizeram necessárias dentre elas, a utilização de uma malha fixa tridimensional com 67 x 43 x 8 células para mapear a região de interesse em Angra dos Reis, um espaço de 16.750 x 10.750 x 1.700 metros.

Devido a necessidade desta configuração de malha, o espaço ocupado por cada célula representa um volume considerável (baixa resolução espacial ou malha grossa) e, quando utilizado pelo SCA para o cálculo do campo de velocidade do vento o resultado obtido representa para cada célula um vetor velocidade do vento de iguais proporções ao tamanho da célula.

Tal resolução espacial gera campos de velocidade de vento com baixa resolução, que como consequência direta, a complexidade do terreno como prédios, residências e principalmente pequenas variações de altura do terreno que normalmente gerariam dispersões no vetor velocidade do vento são simplesmente desprezados.

Para que a trajetória da nuvem radioativa no caso de um acidente nuclear seja a mais precisa possível, é necessário que o campo de vento também o seja e, a complexidade do terreno não deve simplesmente ser ignorada. Com uma representação mais refinada do terreno, é possível se obter uma melhor previsão da dispersão de radionuclídeos no meio, possibilitando um melhor planejamento da evacuação de pessoas, visando uma minimização na exposição das mesmas.

Mesmo com os atuais processadores existentes no mercado, um pequeno aumento na resolução espacial no SCA, aumenta significativamente o custo computacional do

sistema que recai no problema onde programas concebidos para execução sequencial (em um único processador) ainda se tornam lentos. Desta forma, entende-se que uma arquitetura computacional que suporte computação paralela seja uma solução eficiente para que o aumento na resolução espacial do SCA seja realizado.

O uso da Computação Paralela vem permitindo a utilização eficiente (em tempos aceitáveis) de modelos mais precisos em simulações de alto custo computacional, que no caso particular do campo de vento, pequenas ondulações no terreno passam a modificar os padrões de dispersão do campo de velocidade de vento.

Trabalhos anteriores (ALMEIDA, 2009), (MORAES, 2012), (ROCHA, 2008), (HUSEMANN, GOBBI, *et al.*, 2013) e (NVIDIA CORPORATION, 2015) mostraram que as simulações baseadas em GPU podem reduzir centenas de vezes o tempo de uma simulação.

Considerando a extrema importância na análise de riscos e tomada de decisão, antes (em tempo de planejamento), durante e depois da ocorrência de incidentes com liberação de material radioativo na atmosfera, em instalações nucleares de potência, existe um atual interesse no aumento da resolução da simulação da dispersão dos radionuclídeos que, para tal, deve-se aumentar a resolução do campo de vento.

1.8. METODOLOGIA

Inicialmente foram estudados artigos e teses relacionados com programação CUDA e campo de vento. Em seguida, foi aprofundado um estudo sobre o código do campo de vento de forma que pudesse ser desenvolvido um aumento da resolução espacial nos eixos x e y próximos a resolução compatível ao de dispersão urbana. Diversos níveis de refinamento (maior resolução espacial) foram utilizados.

Foi feita uma investigação do comportamento do campo de vento com um nível de refinamento maior para uma prévia identificação de possíveis discrepâncias devido ao aumento da resolução espacial. Foi feito um estudo dos possíveis trechos do algoritmo do campo de vento para determinar quais funções seriam paralelizáveis e de alto custo computacional.

Na sequência, foi realizado um estudo, pesquisa e aprofundamento das técnicas de paralelização e programação de GPU para cada trecho do código anteriormente identificado como paralelizável do algoritmo, utilizando CUDA. Nesta etapa, as sub-rotinas de cálculo foram novamente testadas e seus resultados confrontados com aqueles já obtidos anteriormente. Finalmente, o algoritmo foi otimizado utilizando-se a otimização por enxame de partículas (PSO) de forma a gerar um campo de vento tridimensional não divergente paralelo com as melhores configurações encontradas pelo algoritmo PSO, desta forma minimizando o tempo total de execução do algoritmo.

Ao término de cada experimento, os ganhos em termos de tempo de processamento foram comparados e analisados. Durante todo o trabalho, o algoritmo foi submetido a simulações realísticas, utilizando dados reais adquiridos das estações situadas nas torres A, B, C e D conforme ilustrado na Figura 1 pelas respectivas letras. Ao final foi feita uma avaliação qualitativa considerando os trabalhos relacionados comparando os resultados obtidos neste trabalho com os apresentados na literatura atual.

CAPÍTULO 2. FUNDAMENTAÇÃO TEÓRICA

2.1. A ATMOSFERA

A atmosfera é constituída basicamente por uma mistura de gases que variam em quantidade dependendo da região e altitude. Aproximadamente nos primeiros 80 Km esta mistura é bastante homogênea e segundo (LIOU, 1980) é composta basicamente por 78,00% de nitrogênio, 20,90% de oxigênio, 0,90% de argônio, 0,03% de dióxido de carbono e ainda outros gases em quantidades praticamente desprezíveis (abaixo de 0,001%) como por exemplo o neônio, hélio, metano, criptônio dentre outros conforme ilustrado na Tabela 1.

Constituintes	Porcentagem
Nitrogênio (N)	78,00 x 100
Oxigênio (O2)	20,90 x100
Argônio (Ar)	0,90 x 100
Dióxido de carbono (Co2)	0,03 x 100
Neônio (Ne)	18,18 x 10-4
Hélio (He)	5,24 x 10-4
Criptônio (Kr)	1,14 x 10-4
Xenônio (Xe)	0,89 x 10-4
Hidrogênio (H2)	0,50 x 10-4
Metano (CH4)	1,50 x 10-4
Óxido nitroso (N2O)	0,27 x 10-4
Monóxido de carbono (CO)	0,19 x 10-4
Vapor d´água (H2O)*	0,0 - 0,04 x 100
Ozônio (O3)*	0,0 - 12,00 x 10-4
Dióxido de enxofre (So2)*	0,10 x 10-6
Dióxido de nitrogênio (NO2)*	0,10 x 10-6
Amônia (NH3)*	0,40 x 10-6
Óxido nítrico (NO)*	0,50 x 10-7
Sulfito de hidrogênio (H2S)*	0,50 x 10-8

* Constituintes de concentração variável

Tabela 1 - Adaptada de (LIOU, 1980)

A área de interesse deste trabalho se encontra na primeira camada atmosférica chamada de troposfera ou baixa atmosfera que se estende do solo até aproximadamente 10Km de altura, mais precisamente em uma região chamada de camada limite atmosférica, camada limite planetária ou ainda baixa troposfera, que é uma região próxima da superfície que varia de 200 m a 2.000 m dependendo da localização e condições de estabilidade (STULL, 1988).

A camada limite atmosférica ou CLA pode ser dividida verticalmente em 3 outras regiões: a subcamada laminar, a camada superficial e a camada de transição. A sub camada laminar é a camada mais próxima do solo e se estende da superfície até a altura da rugosidade aerodinâmica[3] (z_0), caracterizada por um escoamento laminar logo acima da superfície (aproximadamente 1mm) e um escoamento turbulento não totalmente desenvolvido no restante desta camada. A camada superficial é a camada intermediária da camada limite atmosférica, onde os fluxos turbulentos são aproximadamente constantes com a altura. Esta camada se estende desde a altura da rugosidade dinâmica (z_0) até aproximadamente 100m.

A camada de transição é a camada mais distante do solo, que ainda sofre os efeitos térmicos e mecânicos e se estende do final da camada superficial até o final da camada limite atmosférica. Esta camada possui características e nomes próprios dependendo da

---

[3] Formalmente, corresponde à altura a partir do solo (em metros) onde a velocidade do vento é igual a zero.

estabilidade atmosférica. A estabilidade atmosférica é uma medida do grau de turbulência na atmosfera, que é gerada pela turbulência mecânica e pela turbulência térmica.

Dependendo da hora do dia, temos uma maior ou menor atuação da turbulência térmica sobre a turbulência mecânica, desta forma, durante o dia com a temperatura na atmosfera decrescendo temos grandes movimentos verticais é a chamada condição instável de estabilidade. Por outro lado, no período noturno temos a temperatura na atmosfera crescendo e poucos movimentos verticais, é a chamada condição estável de estabilidade. Nas transições entre o dia e a noite, podemos observar que não há interferência de temperatura nos movimentos verticais, é a chamada condição neutra de estabilidade.

O transporte dos radionuclídeos na atmosfera lançados de uma fonte, podem ocorrer através da advecção que é o transporte provocado pela velocidade do vento e é considerado o principal mecanismo de dispersão, a difusão turbulenta que é o processo pelo qual os radionuclídeos são transportados devido ao movimento turbulento na atmosfera e a difusão molecular, que é o transporte devido ao movimento térmico das moléculas de todas as partículas a temperatura acima do zero absoluto. A difusão molecular ocorre em níveis microscópicos e pode ser facilmente desprezado se comparados com os mecanismos de advecção e difusão turbulenta.

A estabilidade atmosférica é uma medida do grau de turbulência e reflete as condições de dispersão da atmosfera. Segundo (ZANNETTI, 1990), a estabilidade atmosférica pode ser determinada através:

- De métodos empíricos (classe de estabilidade de Pasquill)
- Do número de Richardson fluxo (R_f)

- Do número de Richardson gradiente (R_i)
- Do comprimento de Monim Obukhov (L)

No presente trabalho, a condição de estabilidade atmosférica é efetuada com base nas classes de estabilidade de Pasquill (PASQUILL, 1961) a fim de incorporar a turbulência induzida nas componentes da velocidade do vento.

2.2 MODELOS DE DISPERSÃO ATMOSFÉRICAS DE RADIONUCLÍDEOS

Após os poluentes serem lançados na atmosfera, os mesmos são governados por processos de transporte e difusão, tais processos são numerosos e de uma complexidade tal que para uma correta descrição é necessário a utilização de modelos matemáticos, pois sem estes, seria praticamente impossível a descrição do comportamento dos poluentes na atmosfera. Esses modelos procuram resolver as equações fundamentais de transporte aplicados a condições iniciais e de contorno de forma a obter uma possível solução do comportamento dos poluentes na atmosfera (MELO, 2011).

Os modelos matemáticos utilizados na dispersão atmosférica de uma forma geral, podem ser classificados em duas principais classes, os eurelianos e os lagrangeanos. Os modelos eurelianos utilizam uma aproximação da equação de conservação de massa com uma referência fixa em relação a terra (SOARES, 2010), já os modelos lagrangeanos baseiam-se nas trajetórias para o movimento das partículas do fluido utilizando um referencial móvel que se desloca com a pluma de poluentes (MORAES, 2001). Podemos ainda destacar os modelos gaussianos que podem ser considerados como uma subclasse dos dois modelos anteriormente citados, que foram obtidos pela consideração da homogeneidade da turbulência e do vento e da simplificação da equação de transporte de massa (LONGHETTO, 1980).

Diversos modelos matemáticos são utilizados para a modelagem dos poluentes atmosféricos e como consequência, a dispersão atmosférica de radionuclídeos. Podemos por exemplo citar os dois modelos mais utilizados que são o modelo AERMOD (AERmic MODel) (CIMORELLI, PERRY, *et al.*, 2004), que é um modelo de dispersão gaussiano de pluma de estado estacionário, que incorpora a dispersão do ar na direção vertical e horizontal com base na estrutura da turbulência da camada limite planetária, incluindo o tratamento de fontes superficiais e elevadas (YANG, YANG, *et al.*, 2005) e o modelo de dispersão de bufadas CALPUFF (*California Puff Model*) (SCIRE, STRIMAITIS e YAMARTINO, 2000), que é um modelo lagrangeano não estacionário de dispersão onde a pluma é representada por uma série de bufadas (puffs), de material poluente que simula os efeitos de condições meteorológicas variáveis no tempo e no espaço sobre o transporte, transformação e remoção dos poluentes.

Atualmente o modelo de dispersão atmosférica utilizado nas centrais nucleares brasileiras, é um modelo gaussiano. Modelos gaussianos, na prática apenas conseguem ser utilizados em terrenos que possuem sua geografia plana e se utilizam das classes de estabilidade de Pasquill para determinar de forma empírica os coeficientes de dispersão (ZANNETTI, 1990), (TILL e GROGAN, 2008), (SEINFELD e PANDIS, 2006). O modelo de dispersão atmosférica urbana (Urban Dispersion Model – UDM), é um modelo matemático do tipo gaussiano com dispersão de bufadas, que consiste em representar através de equações, a dispersão atmosférica de poluentes em áreas urbanas e ao longo dos anos, vem ganhando grande interesse devido a presença de centrais nucleares próximas de regiões urbanas, como é o caso da CNAAA.

2.2.1. O Modelo de Dispersão Atmosférica Urbana – UDM

Na análise do vento associada à ocupação urbana, existe uma superfície rugosa determinante do comportamento das camadas do ar próximas aos edifícios e acima das coberturas dos prédios. Desta forma, para modelar o comportamento dos ventos urbanos, é fundamental o conhecimento das características aerodinâmicas das cidades e do relevo (GRIMMOND e OKE, 1999).

Na camada limite atmosférica (CLA) (planetary boundary layer - PBL) que é a camada de interesse para a análise dos ventos urbanos, o escoamento do fluxo de ar possui comportamento diferenciado ao longo de sua extensão vertical devido a influência de pequenas elevações do terreno e de construções como casas e prédios. Seu comportamento varia do turbulento nos níveis mais próximos do solo ao não turbulento no topo da CLA.

Oke (OKE, 1978) propôs a divisão da CLA em dois níveis: a "camada urbana ao nível das coberturas" (urban canopy layer - UCL) que abrange a extensão do solo até a altura média das coberturas dos edifícios e a "camada limite urbana" (urban boundary layer - UBL) que é definida como a camada adjacente à superfície do solo que se estende até o nível onde a influência do atrito é nula. Na Figura 4 podemos notar as camadas verticais nas áreas urbanas propostas por Oke.

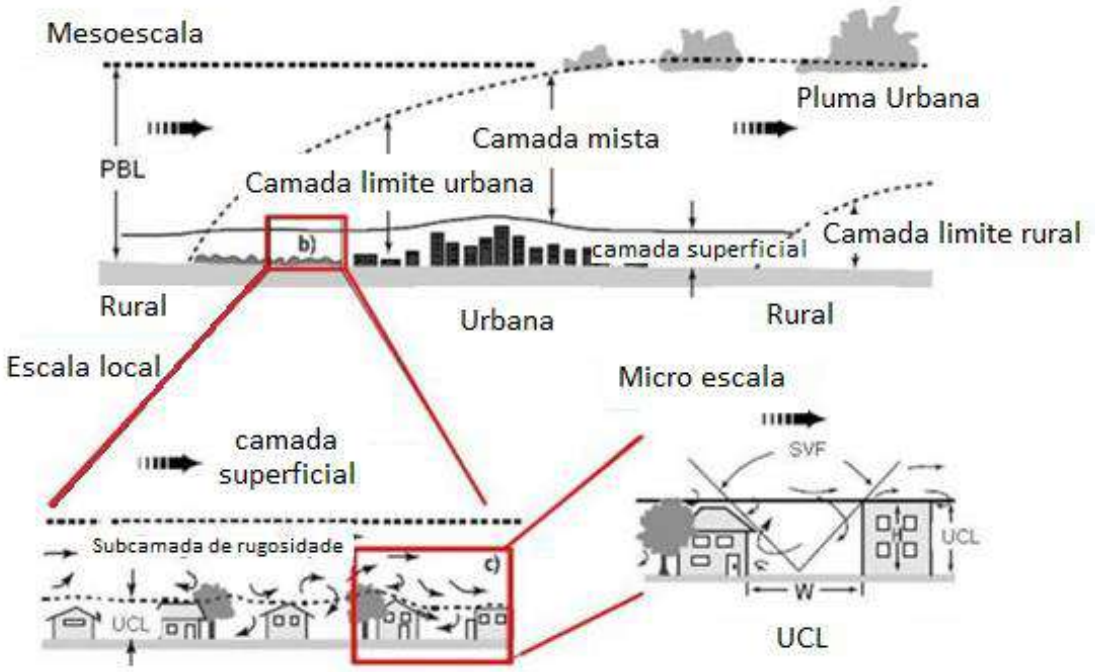

Figura 4 – Camadas verticais nas áreas urbanas
Adaptado de (OKE, 1978)

Segundo (HALL, SPANTON, *et al.*), o modelo de dispersão urbana (urban dispersion model - UDM) foi desenvolvido para estimar a dispersão de bufadas (puffs) de contaminantes do ar a curtas distâncias em áreas urbanas, onde as obstruções da superfície, principalmente edifícios e pequenas alterações no relevo podem modificar os padrões de dispersão.

O modelo foi projetado para lidar com distâncias entre cerca de 10m e 10 km; além desta distância, os dispersantes tendem a encher toda a camada limite e a natureza da superfície torna-se menos importante para a dispersão. Segundo (PEDERSEN, LEACH e HANSEN, 2007) em seu relatório Biological Incident Response: Assessment of Airborne Dispersion, o tamanho da célula deve ser menor que, preferencialmente, 100 metros quadrados para áreas urbanas.

2.3. SISTEMA DE CONTROLE AMBIENTAL – SCA

O Sistema de Controle Ambiental (SCA), que é um sistema de previsão da dispersão atmosférica de radionuclídeos (DAR), utilizado na Central Nuclear Almirante Álvaro Alberto (CNAAA) e possui como objetivo primordial, servir de ferramenta para o auxílio na tomada de decisão, caso haja a necessidade de se realizar uma evacuação, em virtude de um possível acidente nuclear com liberação de radionuclídeos para o meio ambiente. A estrutura modular do sistema SCA usado na CNAAA, é composto por 4 módulos principais: Termo de Fonte, Campo de Vento, Dispersão de Pluma e Módulos de Projeção, como mostra a Figura 5.

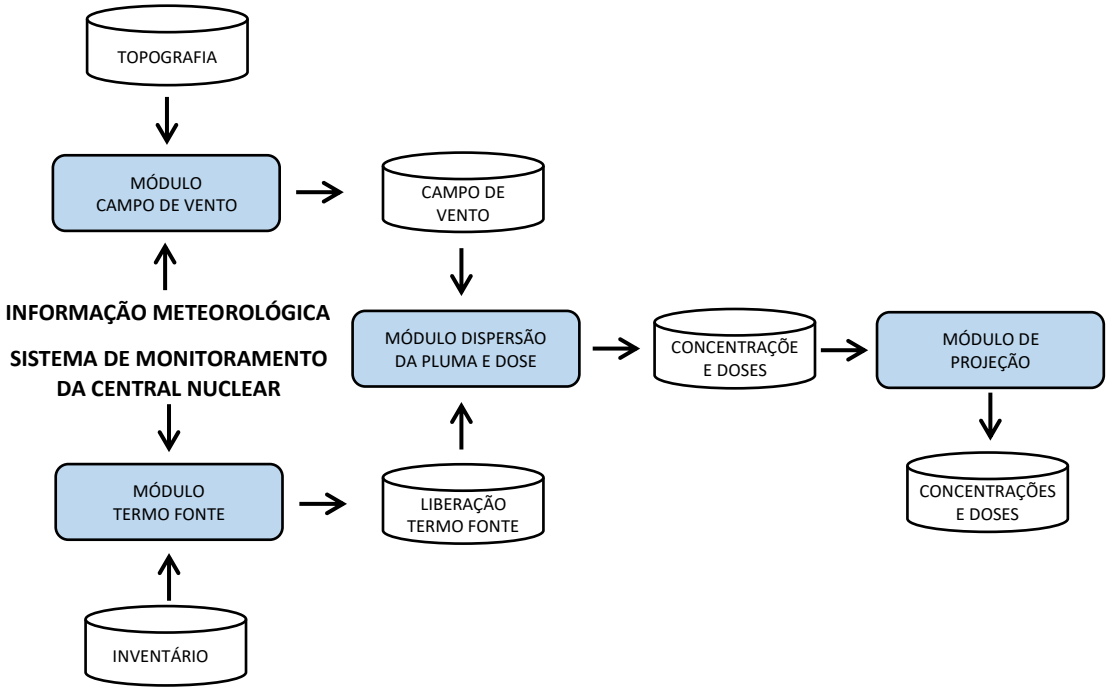

Figura 5 – Estrutura modular do Sistema de Controle Ambiental (SCA)

Em situações de emergência o fator tempo é primordial, as tomadas de decisões devem ser feitas de forma rápida e precisa, tendo como base informações confiáveis. O SCA passa aos tomadores de decisões, informações importantes ao que se refere a situação atual, em termos das consequências para a população abrangida pela área

delimitada pelo sistema. Trata-se de um sistema que deve ser usado fundamentalmente em situações de emergência ou para simulação de treinamento destas situações. O SCA pode ser ativado de duas maneiras: de forma automática e de forma manual. Na forma automática o SCA é ativado no modo de operação em emergência toda vez que que uma das seguintes situações ocorrerem:

- Sinal de desarme do reator;
- Sinal de injeção de segurança;
- Alarme de alto nível de radiação em qualquer um dos monitores de radiação;
- Alarme de alta atividade nas salas dos compressores do sistema de tratamento de rejeitos gasosos;
- Baixa pressão nos tanques do sistema de tratamento de rejeitos gasosos;
- Alta pressão no tanque de alivio do pressurizador (pressão de ruptura do disco)

Já na ativação manual, o SCA pode ser ativado tanto no modo de operação de emergência, no caso de um acidente real ou no modo de operação de simulação, para treinamento e/ou análise. No modo de operação em emergência, é considerado a existência de um acidente real e só pode ser desativado pelo pessoal da Proteção Radiológica, com ordem expressa da Gerência de Situações de Emergência (FURNAS CENTRAIS ELÉTRICAS S.A., 1987). Já no caso do modo de operação em simulação, o sistema simula um acidente, mas pode ser desativado a qualquer tempo. O objetivo do SCA, consiste na avaliação da dose radiológica no meio ambiente e o cálculo da dose e da previsão de dose recebida por um indivíduo em um determinado ponto afastado da

fonte geradora de radiação que, no caso de um acidente nuclear, seria a própria usina nuclear.

Para que a dose radiológica neste ponto seja devidamente calculada, é necessário determinar como a concentração dos radionuclídeos liberados no acidente variam espacialmente e temporalmente na atmosfera. Por sua vez, para computo da distribuição espacial e da evolução temporal do processo de transporte dos radionuclídeos emitidos, é necessário que seja calculada a quantidade de material radioativo liberado durante o acidente e como foi o transporte e difusão deste material desde a sua origem até o seu destino.

Como o transporte e difusão ocorrem em meio atmosférico é necessário levarmos em consideração as variações com o tempo das condições meteorológicas como a velocidade e a estabilidade do vento. Atualmente o SCA leva em consideração um intervalo de tempo de 15 minutos denominado de ciclo. Em cada ciclo, as condições meteorológicas são mantidas constantes para que sejam efetuados os cálculos necessários para a avaliação da dose radiológica em um determinado ponto,

A cada ciclo do sistema os seguintes cálculos são realizados:

1. Avaliação da taxa de liberação de radionuclídeos para a atmosfera;
2. Determinação do campo de velocidade e estabilidade do vento;
3. Cálculo do transporte e difusão do material radioativo na atmosfera;
4. Avaliação da distribuição espacial da dose; e
5. Cálculo da dose decorrente de projeções do ciclo atual.

A avaliação da taxa de liberação dos radionuclídeos para a atmosfera, depende fundamentalmente do tipo de acidente (real ou simulado), cada acidente em particular

possui suas próprias características que interferem na quantidade dos radionuclídeos liberados. A atividade inicial de cada radionuclídeo disponível para liberação depende do acidente ocorrido e do tipo de radionuclídeo considerado identificando-se os caminhos pelos quais essa liberação pode ocorrer.

Por conta destas particularidades, o operador da usina ao diagnosticar/simular o acidente ocorrido, deve informar ao sistema através de uma linha de comando qual o acidente em questão para que a atividade total liberada no ciclo possa ser calculada pelo modelo de termo fonte, fazendo um balanço da atividade através de cada caminho possível compatível com o acidente, de acordo com o status dos diversos componentes dos caminhos.

A taxa de liberação média do ciclo é calculada, dividindo-se a atividade total liberada no ciclo, pela quantidade de bufadas liberadas no mesmo ciclo. O valor da taxa de liberação média é tomado como a atividade liberada por uma bufada no ciclo. Uma vez calculada a taxa de liberação média por ciclo, o SCA deve então calcular a distribuição espacial do campo de velocidade do vento baseado nos valores médios do ciclo obtidos nas torres meteorológicas, da velocidade do vento (magnitude e direção). Para tanto, a topologia da região ocupa uma malha fixa tridimensional de 67 x 43 x 8.

O campo de vento é tornado consistente em massa após um processo de extrapolação e interpolação a partir dos valores fornecidos, através de um processo interativo que elimina as divergências do campo interpolado. Com o campo de vento devidamente conhecido pelo processo acima descrito e distribuído na malha tridimensional, o cálculo do transporte e da difusão dos radionuclídeos na atmosfera é então iniciado e possui um modelo de bufadas tridimensional com trajetória lagrangeana variável e difusão gaussiana.

Cada bufada possui uma taxa de liberação média que foi calculada pelo modulo termo fonte e é liberada e transportada durante um intervalo de advecção, cujo valor é definido a cada ciclo, com a velocidade local do campo de vento no ponto que corresponde ao início do intervalo de advecção. A contribuição de uma bufada para a concentração em cada ponto da malha é determinada pelo modelo gaussiano ao longo da trajetória percorrida pela bufada durante o ciclo e a concentração em cada ponto da malha é o somatório das contribuições médias das bufadas naquele determinado ponto, durante o ciclo que estiver sendo processado.

Os seguintes efeitos físicos são considerados no modelo de transporte e difusão (COPPE/UFRJ - NUCLEAR, LABORATÓRIO DE ANÁLISE E SEGURANÇA, 1987).

- Efeito de esteira (“building wake”) causado pelos edifícios da usina;
- Elevação da pluma (empuxo térmico e/ou quantidade de movimento devido à velocidade de saída do material liberado);
- Depleção seca (deposição de particulados e iodos no solo);
- Depleção molhada (deposição de particulados e iodos em caso de chuva durante o transporte atmosférico);
- Decaimento radioativo;
- Reflexão no solo;
- Variação do coeficiente de dispersão quando a trajetória da bufada atravessa regiões com diferentes classes de estabilidade.

Ao final de cada ciclo, são geradas tabelas que possuem a concentração média de iodos e particulados a nível do solo e para gases nobres a nível do solo, 50m, 125m e 275m. Com base nas tabelas geradas, são calculadas a taxa de dose média no ciclo e as doses acumuladas desde o início do acidente para a tireoide e pulmão, devido à inalação

de iodos, particulados e gases nobres e para o corpo inteiro devido à imersão na nuvem onde somente existe a contribuição dos gases nobres usando-se o modelo de pluma finita.

Assumindo a persistência das condições do ciclo atual, tais como condições meteorológicas e taxa de liberação dos radionuclídeos para a atmosfera, o SCA fornece projeções para uma e duas horas, para as taxas de dose média e doses acumuladas, contados a partir do final do ciclo atual.

2.4. O MÓDULO CAMPO DE VENTO

A abordagem gaussiana de modelos de campo de vento, muito utilizada devido a sua simplicidade, tem sua aplicabilidade bastante restringida e não são na pratica utilizadas uma vez que estas abordagens são desenvolvidas com base em hipóteses simplificadoras bastante restritivas (BROWN, ARYA e SNYDER, 1993), tais como campo de ventos uniforme e constante com o tempo e relevo plano. Os modelos gaussianos na prática, apenas conseguem ser utilizados em terrenos que possuem sua geografia plana e com praticamente nenhuma ocupação do solo (FINARDI, TINARELLI, *et al.*, 1998).

Na prática, devido à complexidade do terreno tais como a ocupação do solo por prédios e residências e ventos não uniformes, estes geram campos de ventos com escoamentos locais e como consequência, dispersões particulares em cada local que não conseguem ser reproduzidas por modelos gaussianos, devido as simplificações utilizadas nos mesmos.

Modelos de dispersão atmosférica mais sofisticados normalmente requerem como entrada um campo de vento tridimensional da região de interesse. Desta forma, em regiões complexas como áreas urbanas, metodologias devem ser utilizadas para a obtenção do

campo de vento tridimensional, como um passo inicial para o estudo da dispersão de radionuclídeos. (FINARDI, TINARELLI, *et al.*, 1998) Apresentaram uma revisão completa de metodologias para obtenção de campo de vento em condições de escoamentos complexos.

A metodologia usada para gerar o campo de velocidade de vento do SCA está baseada no modelo WEST (Winds Extrapolated from Stability and Terrain) (FABRICK, SKLAREW e WILSON, 1977) implementado no módulo que calcula o campo de vento. Para descrever a distribuição espacial do vento na região de interesse, usa-se um campo de velocidade tridimensional e não divergente, ou seja, após um processo de extrapolação e interpolação a partir dos valores fornecidos pelas torres, o campo de vento é tornado consistente em massa, através de um processo iterativo de eliminação das divergências do campo interpolado. (COPPE/UFRJ - NUCLEAR, LABORATÓRIO DE ANÁLISE E SEGURANÇA, 1987, p. 3-1-2 e 3-3-1).

A presente versão do módulo responsável pelo cálculo do campo de vento é resultado da adaptação, pelo PEN-COPPE/UFRJ, do programa west.furnas-1p, que resultou da adaptação, pela NUS Corporation, do modulo WEST (Winds Extrapolated from Stability and Terrain) implementado no código SMOG (Simulation Model of Ozone Generation), da *California Air Resources Board*. A região de interesse é representada por um paralelepípedo cuja a base inferior está apoiada no ponto topográfico mais baixo da região conforme ilustrado na Figura 6.

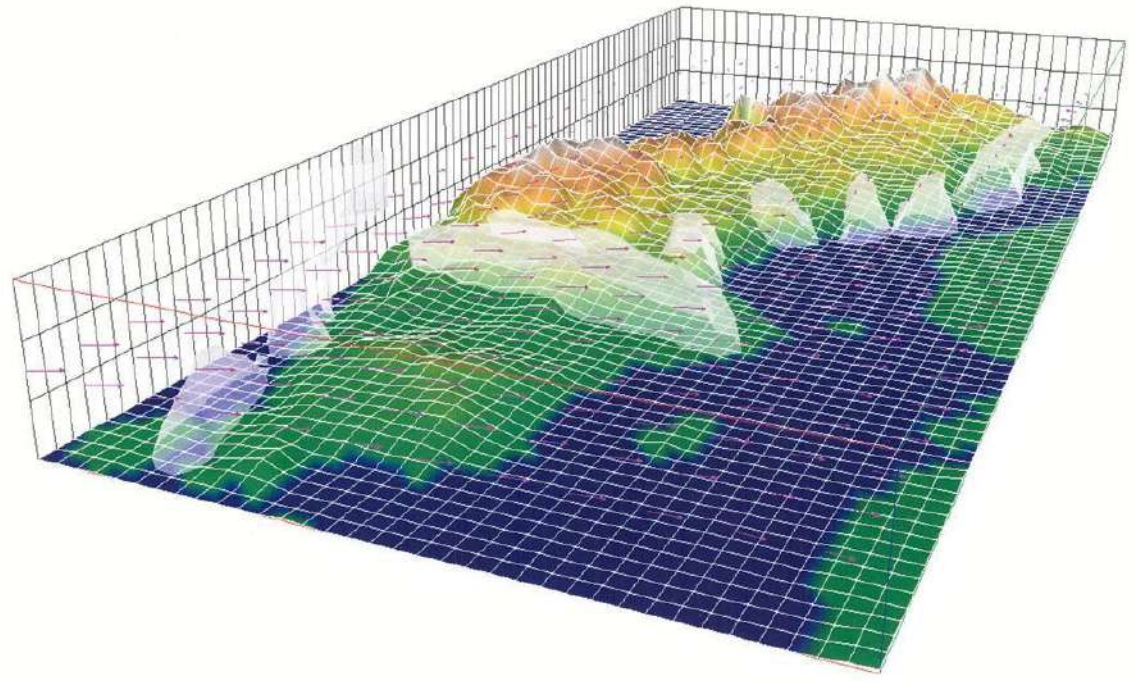

Figura 6 – Esquema do reticulado tridimensional
Fonte: Adaptado de http://www.smhi.se/en/research/research-departments/air-quality/match-transport-and-chemistry-model-1.6831, acessado em 19/02/2015

As dimensões do paralelepípedo foram determinadas pelas limitações de armazenamento dos computadores disponíveis na época sendo seu volume subdividido em outros volumes retangulares com dimensões $\Delta x_n = \Delta y_n = 250$ m e Δz_k = variável, conforme ilustrado na Tabela 2.

Z_8	1700 m	ΔZ_8	450 m
Z_7	1250 m	ΔZ_7	450 m
Z_6	800 m	ΔZ_6	350 m
Z_5	450 m	ΔZ_5	200 m
Z_4	250 m	ΔZ_4	100 m
Z_3	150 m	ΔZ_3	50 m
Z_2	100 m	ΔZ_2	50 m
Z_1	50 m	ΔZ_1	50 m
Z_0	0 m		

Tabela 2 – Níveis de altura do paralelepípedo

Esta configuração caracteriza um reticulado tridimensional fixo de tamanho 67 x 43 x 8, composto de células retangulares com dimensões Δx, Δy e Δz_k como pode ser observado na Figura 7.

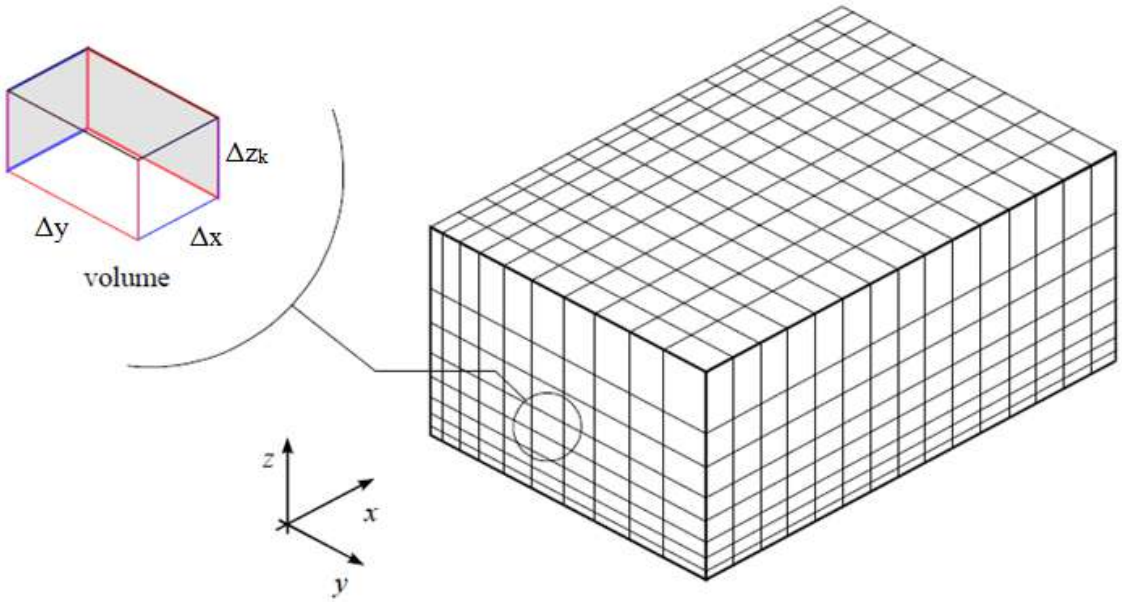

Figura 7 – Células do reticulado tridimensional

No sistema de coordenadas retangulares cartesianas utilizado, a coordenada X está orientada na direção Leste-Oeste, aumentando no sentido Leste. A coordenada Y está orientada na direção Norte-Sul, aumentando no sentido Norte e a coordenada Z é a direção vertical, aumentando no sentido do topo do paralelepípedo (COPPE/UFRJ - NUCLEAR, LABORATÓRIO DE ANÁLISE E SEGURANÇA, 1987, p. 3-3-2).

A topografia do terreno é representada por células obstáculo ou seja, as células sofrem uma elevação na coordenada Z de forma a corresponder com a elevação do terreno (Tabela 2). A Figura 8 nos mostra uma representação de uma topografia simples onde podemos visualizar as cotas de 150m, 100m 50m e 0m.

O número de células obstáculo (coordenada Z) em cada posição (X, Y) da grade reticulada, é dado como parâmetro de entrada que contém o número de células correspondentes às elevações do terreno na posição considerada.

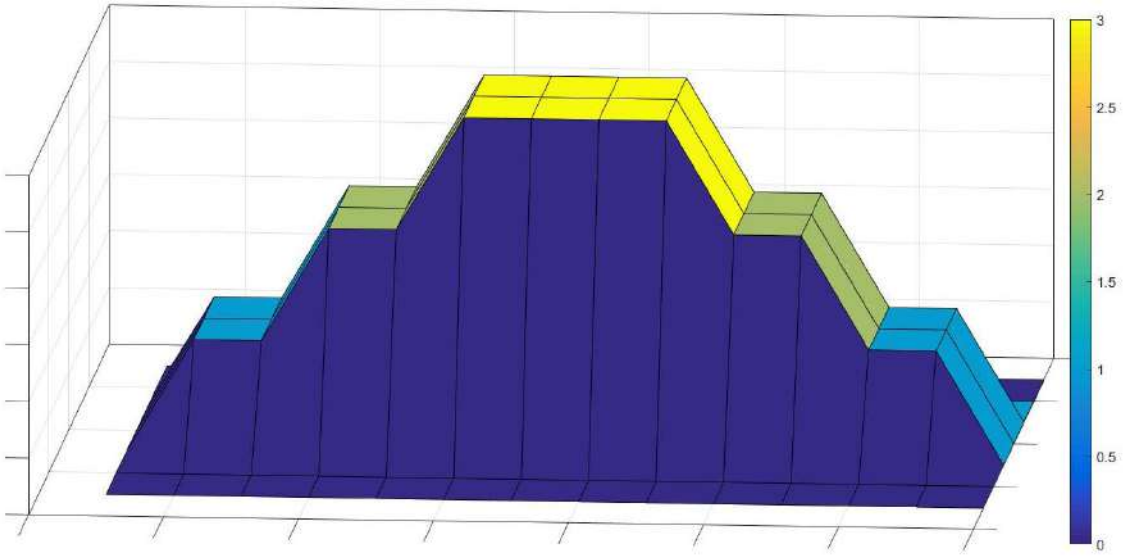

Figura 8 – Células obstáculo

A formulação matemática para a produzir um campo de velocidade de vento tridimensional e não divergente utilizando o modelo de campo de vento Winds Extrapolated from Stability and Terrain é visto na próxima seção mas, pode ser resumidamente descrito pelas seguintes etapas (COPPE/UFRJ - NUCLEAR, LABORATÓRIO DE ANÁLISE E SEGURANÇA, 1987, p. 3-3-7 à 3-3-17):

1. As medidas de velocidade, módulo e direção do vento no plano horizontal são fornecidas por 4 torres meteorológicas como dados de entrada;
2. Estas mesmas torres fornecem dados sobre a condição de estabilidade atmosférica;
3. Os valores observados nas torres, são extrapolados ao longo das demais células da região de interesse;

4. As componentes horizontais de velocidade u e v são então determinadas em todas as células;
5. Um campo de velocidade é gerado fazendo-se uma interpolação das velocidades;
6. O campo de velocidade obtido é ajustado, a fim de que seja satisfeita a condição de não divergência
7. Todas as divergências residuais são adicionadas as componentes verticais por um processo iterativo.

2.5. O MODELO DO CAMPO DE VENTO

O modelo de campo de vento utilizado neste trabalho é o Winds Extrapolated from Stability and Terrain (WEST) que é um modelo de campo de vento de diagnóstico que calcula campos de vento tridimensionais, dependentes do terreno e sem divergência, com base nos dados de vento observados como entrada.

A região de interesse (domínio computacional) é discretizada em uma grade tridimensional. A fim de produzir uma primeira estimativa do campo do vento, extrapolações e interpolações do vento observado são aplicadas a todas as células no domínio computacional. Em seguida, o campo de vento estimado é iterativamente ajustado para se tornar não divergente, representando os efeitos do terreno e da estabilidade atmosférica.

O objetivo é determinar, a partir dos componentes u^0, v^0 e w^0 do campo de vento estimado, uma componente de vento ajustada u, v e w que satisfaça a condição de divergência nula (equação (1)) para cada célula de toda a região de interesse.

$$D = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0$$

(1)

Onde *D* é o divergente e *u*, *v* e *w* são as componentes da velocidade do vento nas direções x, y e z, respectivamente. Os componentes ajustados das velocidades não divergentes são do tipo:

$$\begin{cases} u = u^0 + \tau_X \dfrac{\partial \phi}{\partial x} \\ v = v^0 + \tau_Y \dfrac{\partial \phi}{\partial y} \\ w = w^0 + \tau_Z \dfrac{\partial \phi}{\partial z} \end{cases}$$

(2)

Onde ϕ é o potencial de perturbação da velocidade e τ_X, τ_Y e τ_W são coeficientes de transmissão baseados em perfis de temperatura obtidos a partir de sondagens aéreas superiores (valores podem ser vistos em (FABRICK, SKLAREW e WILSON, 1977)).

Substituindo (2) em (1) leva a equação de Poisson para o potencial de perturbação da velocidade:

$$D = \left[\frac{\partial}{\partial x}\left(\tau_X \frac{\partial \phi}{\partial x} \right) + \frac{\partial}{\partial y}\left(\tau_Y \frac{\partial \phi}{\partial y} \right) + \frac{\partial}{\partial z}\left(\tau_Z \frac{\partial \phi}{\partial z} \right) \right] + \left[\frac{\partial u^0}{\partial x} + \frac{\partial v^0}{\partial y} + \frac{\partial w^0}{\partial z} \right] = 0$$

(3)

Para determinar $\dfrac{\partial D}{\partial \phi}$, considere uma grade tridimensional, na qual as componentes de velocidade estão centrados nas faces das células, enquanto o potencial de perturbação da velocidade é centrado na célula. A equação das diferenças finitas para o divergente é, então:

$$D_{i,j,k} = \frac{1}{\delta x}\left(u_{i+\frac{1}{2},j,k} - u_{i-\frac{1}{2},j,k} \right) + \frac{1}{\delta y}\left(v_{i,j+\frac{1}{2},k} - v_{i,j-\frac{1}{2},k} \right) + \frac{1}{\delta z}\left(w_{i,j,k+\frac{1}{2}} - w_{i,j,k-\frac{1}{2}} \right) = 0$$

(4)

Aplicando a regra de cadeia:

$$\frac{\partial D_{i,j,k}}{\partial \phi_{i,j,k}} = \frac{1}{\delta x}\left(\frac{\partial u_{i+\frac{1}{2},j,k}}{\partial \phi_{i,j,k}} - \frac{\partial u_{i-\frac{1}{2},j,k}}{\partial \phi_{i,j,k}}\right) + \frac{1}{\delta y}\left(\frac{\partial v_{i,j+\frac{1}{2},k}}{\partial \phi_{i,j,k}} - \frac{\partial v_{i,j-\frac{1}{2},k}}{\partial \phi_{i,j,k}}\right) + \frac{1}{\delta z}\left(\frac{\partial w_{i,j,k+\frac{1}{2}}}{\partial \phi_{i,j,k}} - \frac{\partial w_{i,j,k-\frac{1}{2}}}{\partial \phi_{i,j,k}}\right) = 0$$

(5)

Considerando que as componentes de velocidade são da forma descrita na equação (2), as seguintes equações de diferenças finitas podem ser escritas:

$$u_{i+\frac{1}{2},j,k} = u^0_{i+\frac{1}{2},j,k} + \tau_{X_{i+\frac{1}{2},j,k}} \frac{1}{\partial x}\left(\phi_{i+1,j,k} - \phi_{i,j,k}\right)$$

(6)

$$u_{i-\frac{1}{2},j,k} = u^0_{i-\frac{1}{2},j,k} + \tau_{X_{i-\frac{1}{2},j,k}} \frac{1}{\partial x}\left(\phi_{i,j,k} - \phi_{i-1,j,k}\right)$$

(7)

$$v_{i,j+\frac{1}{2},k} = v^0_{i,,j+\frac{1}{2},k} + \tau_{Y_{i,j+\frac{1}{2},k}} \frac{1}{\partial y}\left(\phi_{i,j+1,k} - \phi_{i,j,k}\right)$$

(8)

$$v_{i,j-\frac{1}{2},k} = v^0_{i,,j-\frac{1}{2},k} + \tau_{Y_{i,j-\frac{1}{2},k}} \frac{1}{\partial y}\left(\phi_{i,j,k} - \phi_{i,j-1,k}\right)$$

(9)

$$w_{i,j,k+\frac{1}{2}} = w^0_{i,,j,k+\frac{1}{2}} + \tau_{Z_{i,j,k}+\frac{1}{2}} \frac{1}{\partial z}\left(\phi_{i,j,k+1} - \phi_{i,j,k}\right)$$

(10)

$$w_{i,j,k-\frac{1}{2}} = w^0_{i,,j,k-\frac{1}{2}} + \tau_{Z_{i,j,k}-\frac{1}{2}} \frac{1}{\partial z}\left(\phi_{i,j,k} - \phi_{i,j,k-1}\right)$$

(11)

A partir das equações (6) a (11) pode ser obtido:

$$\frac{\partial u_{i+\frac{1}{2},j,k}}{\partial \phi_{i,j,k}} - \frac{\partial u_{i-\frac{1}{2},j,k}}{\partial \phi_{i,j,k}} = -\frac{\tau_{X_{i+\frac{1}{2},j,k}} + \tau_{X_{i-\frac{1}{2},j,k}}}{\partial x}$$

(12)

$$\frac{\partial v_{i,j+\frac{1}{2},k}}{\partial \phi_{i,j,k}} - \frac{\partial v_{i,j-\frac{1}{2},k}}{\partial \phi_{i,j,k}} = -\frac{\tau_{Y_{i,j+\frac{1}{2},k}} + \tau_{Y_{i,j-\frac{1}{2},k}}}{\partial y}$$

$$\frac{\partial w_{i,j,k+\frac{1}{2}}}{\partial \phi_{i,j,k}} - \frac{\partial w_{i,j,k-\frac{1}{2}}}{\partial \phi_{i,j,k}} = -\frac{\tau_{Z_{i,j,k+\frac{1}{2}}} + \tau_{Z_{i,j,k-\frac{1}{2}}}}{\partial z} \tag{13}$$

(14)

Substituindo (12), (13) e (14) em (5):

$$\frac{\partial D_{i,j,k}}{\partial \phi_{i,j,k}} = -\frac{\left(\tau_{X_{i+\frac{1}{2},j,k}} + \tau_{X_{i-\frac{1}{2},j,k}}\right)\partial y^2 \partial z^2 + \left(\tau_{Y_{i,j+\frac{1}{2},k}} + \tau_{Y_{i,j-\frac{1}{2},k}}\right)\partial x^2 \partial z^2 + \left(\tau_{Z_{i,j,k+\frac{1}{2}}} + \tau_{Z_{i,j,k-\frac{1}{2}}}\right)\partial x^2 \partial y^2}{\partial x^2 \partial y^2 \partial z^2} \tag{15}$$

De acordo com (FABRICK, SKLAREW e WILSON, 1977), fazendo a suposição de que a variação ($\delta\phi_{i,j,k}$) a ser aplicada na perturbação da velocidade ($\phi_{i,j,k}$) é mais fortemente dependente de $\frac{\partial D_{i,j,k}}{\partial \phi_{i,j,k}}$, a seguinte aproximação pode ser escrita:

$$\delta\phi_{i,j,k} \approx \frac{-D_{i,j,k}}{\frac{\partial D_{i,j,k}}{\partial \phi_{i,j,k}}} \tag{16}$$

Substituindo (15) em (16) temos:

$$\delta\phi_{i,j,k} = -\frac{\partial x^2 \partial y^2 \partial z^2}{\left(\tau_{X_{i+\frac{1}{2},j,k}} + \tau_{X_{i-\frac{1}{2},j,k}}\right)\partial y^2 \partial z^2 + \left(\tau_{Y_{i,j+\frac{1}{2},k}} + \tau_{Y_{i,j-\frac{1}{2},k}}\right)\partial x^2 \partial z^2 + \left(\tau_{Z_{i,j,k+\frac{1}{2}}} + \tau_{Z_{i,j,k-\frac{1}{2}}}\right)\partial x^2 \partial y^2} D_{i,j,k} \tag{17}$$

E, finalmente, a correção é aplicada na velocidade, como mostrado nas equações (18) a (23):

$$u_{i+\frac{1}{2},j,k} = u_{i+\frac{1}{2},j,k} + \tau_{X_{i+\frac{1}{2},j,k}} \frac{\delta\phi_{i,j,k}}{\partial x} \tag{18}$$

$$u_{i-\frac{1}{2},j,k} = u_{i-\frac{1}{2},j,k} - \tau_{X_{i-\frac{1}{2},j,k}} \frac{\delta\phi_{i,j,k}}{\partial x} \tag{19}$$

$$v_{i,j+\frac{1}{2},k} = v_{i,j+\frac{1}{2},k} + \tau_{Y_{i,j+\frac{1}{2},k}} \frac{\delta\phi_{i,j,k}}{\partial y} \tag{20}$$

$$v_{i,j-\frac{1}{2},k} = v_{i,j-\frac{1}{2},k} - \tau_{Y_{i,j-\frac{1}{2},k}} \frac{\delta\phi_{i,j,k}}{\partial y} \tag{21}$$

$$w_{i,j,k+\frac{1}{2}} = w_{i,j,k+\frac{1}{2}} + \tau_{Y_{i,j,k+\frac{1}{2}}} \frac{\delta\phi_{i,j,k}}{\partial z} \tag{22}$$

$$w_{i,j,k-\frac{1}{2}} = w_{i,j,k-\frac{1}{2}} - \tau_{Y_{i,j,k-\frac{1}{2}}} \frac{\delta\phi_{i,j,k}}{\partial z} \tag{23}$$

Uma vez que $\delta\phi_{i,j,k}$ é uma mudança local, é necessário propagar os sinais de velocidade para todo o domínio computacional. Assim, dada uma distribuição de velocidade inicial, as seguintes etapas são repetidas até que um critério de convergência seja alcançado:

i) Calcular a divergência usando a equação (4);

ii) Calcular a variação de velocidade de acordo com a equação (17);

iii) Atualização da velocidade de acordo com as equações (18) a (23);

iv) Propagar o sinal de velocidade.

2.6. COMPUTAÇÃO PARALELA

2.6.1. Introdução

A computação paralela é uma forma de computação onde vários cálculos são executados de forma simultânea partindo do princípio que um grande problema

geralmente pode ser resolvido se dividido em problemas menores que então são resolvidos paralelamente. Atualmente existe um grande interesse no assunto pois devido às limitações físicas no aumento da frequência de trabalho dos processadores, a computação paralela se tornou praticamente dominante nas arquiteturas dos computadores atuais sob a forma dos processadores multicore.

Computadores paralelos podem ser classificados em múltiplos computadores e múltiplos processadores. Os computadores paralelos do tipo múltiplos computadores, se utilizam de diversos computadores (que podem possuir processadores multicore cada um) para trabalhar em uma única tarefa como se fossem supercomputadores. Já os computadores paralelos do tipo múltiplos processadores, possuem múltiplos elementos de processamento em somente uma máquina.

Exemplos de computadores paralelos do tipo múltiplos computadores são os clusters, o processamento paralelo massivo, os grids e o MPI e de computadores paralelos do tipo múltiplos processadores são os multicore e GPU. Os clusters (Figura 9) que são formados por um conjunto de computadores, ligados em rede que trabalham como se fossem uma única máquina com grande poder computacional.

Figura 9 – Columbia, o novo (2004) cluster da NASA, 20 Altix clusters executando Linux (10,240 processadores)
Fonte: Adaptado de https://commons.wikimedia.org/wiki/File:Us-nasa-columbia.jpg, acessado em 21/02/2015

O Processamento Paralelo Massivo (massively parallel processing – MPP – Figura 10) é o processamento de um programa em vários processadores. Cada processador se concentra em diferentes partes do programa.

Figura 10 – Processamento paralelo massivo
Fonte: Adaptado de http://www.abapzombie.com/dicas-abap/2012/03/12/processamento-em-paralelo-utilizando-rfc-assincrona, acessado em 21/02/2015

Os Grids (Figura 11) são um modelo computacional que dividem as tarefas entre diversas maquinas, em uma rede local ou na internet, e formam uma máquina virtual. Normalmente as tarefas são executadas quando os usuários não estão utilizando as máquinas. Atualmente o CERN através do CERN OpenLab (www.cern.ch/openlab), se utiliza este tipo de computação paralela.

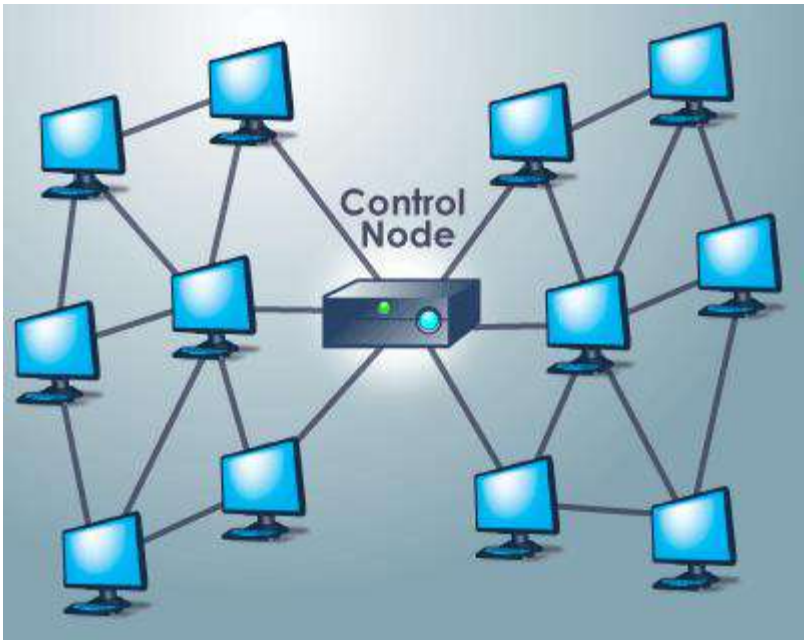

Figura 11 – Processamento em Grids
Fonte: Adaptado de http://computer.howstuffworks.com/grid-computing.htm, acessado em 21/02/2015

O MPI (Figura 12) é um padrão de comunicação de dados e é utilizado para passar informações aos nodos de um cluster. No padrão MPI, uma aplicação normalmente é constituída por um conjunto fixo de processos que se comunicam entre si para executarem diferentes tarefas.

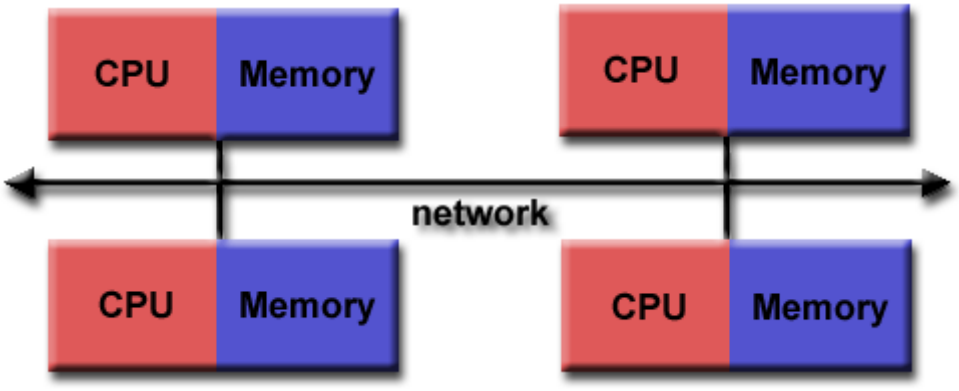

Figura 12 – MPI
Fonte: Adaptado de https://computing.llnl.gov/tutorials/mpi, acessado em 21/02/2015

Na computação paralela do tipo multicore (Figura 13), são colocados diversos núcleos de processamento dentro de um único chip. Os diversos núcleos são responsáveis por dividir as tarefas que normalmente seriam sequenciais entre si e, as executarem de forma paralela.

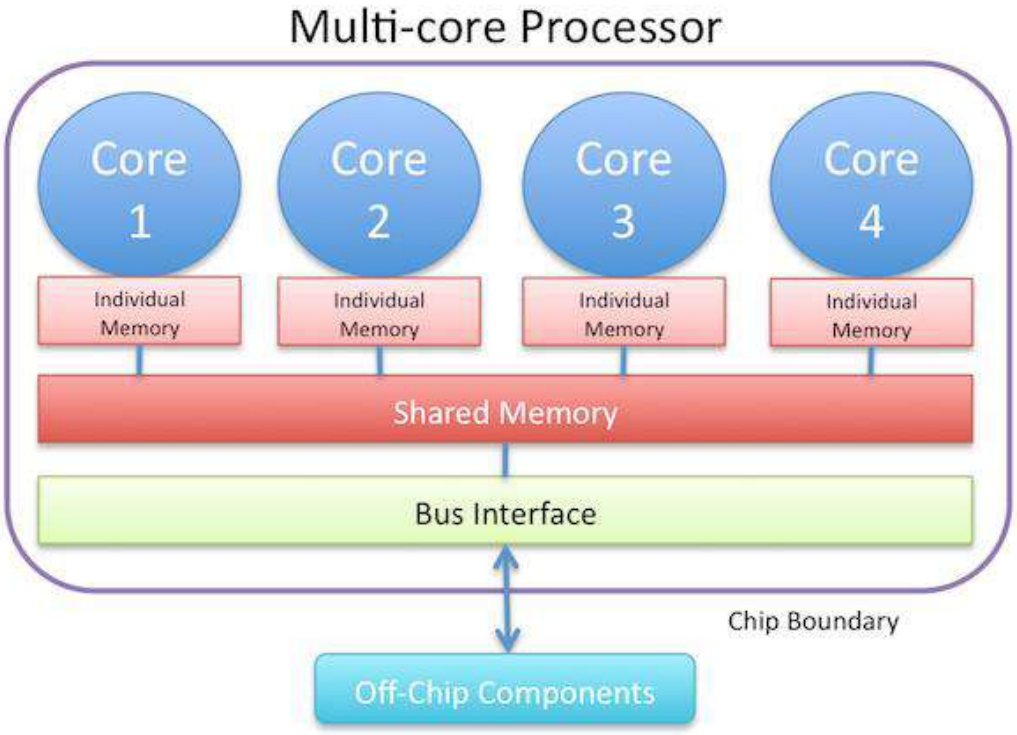

Figura 13 – Multicore
Fonte: Adaptação de http://www.cse.wustl.edu/~jain/cse567-11/ftp/multcore, acessado em 21/02/2015

Na computação paralela com a utilização de GPU (Figura 14) são colocados milhares de processadores dentro de um único chip. As GPUs foram projetadas para executarem milhares de tarefas (threads[4]) em paralelo.

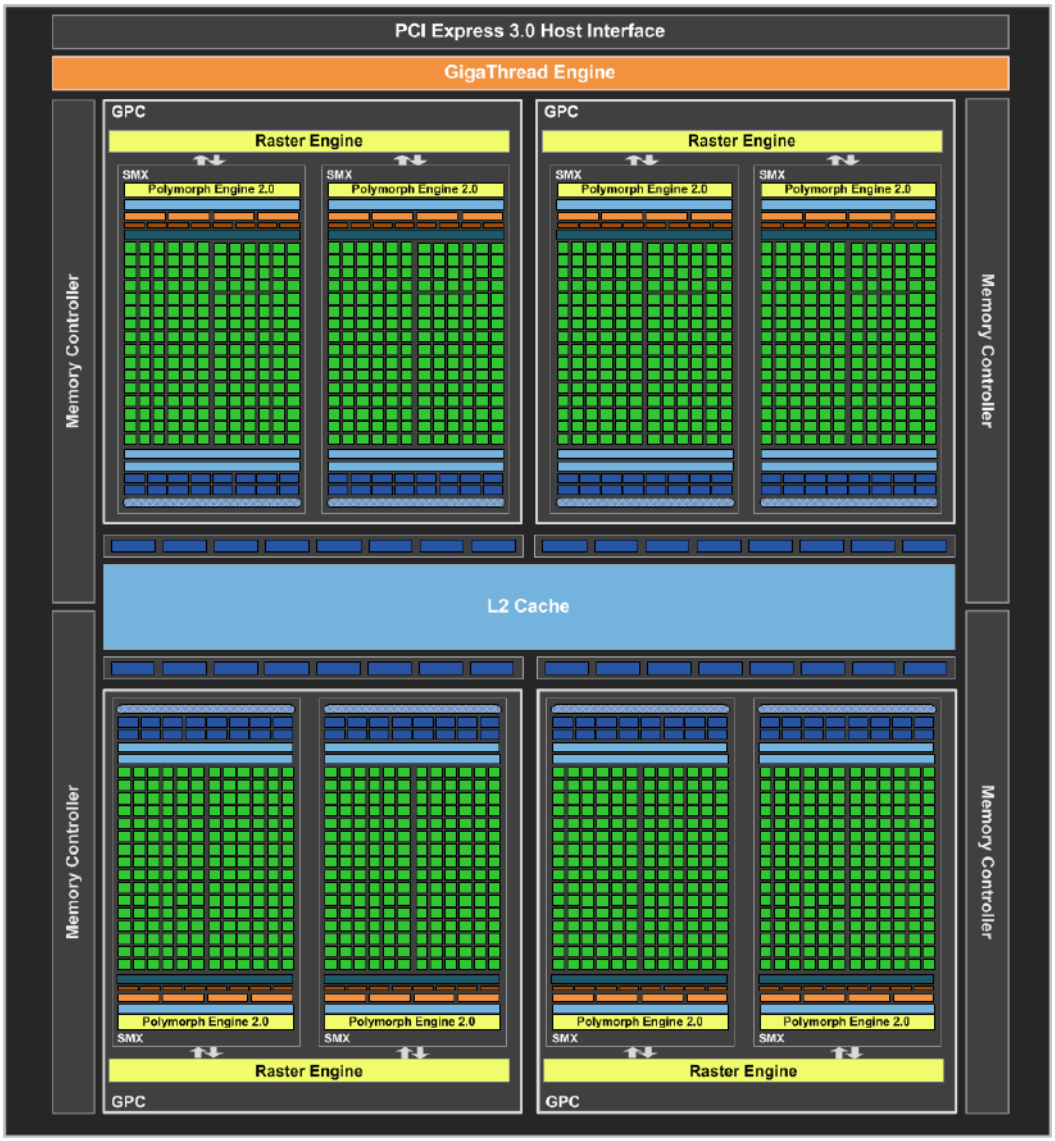

Figura 14 – Diagrama de blocos da GPU GeForce GTX 680
Fonte: (NVIDIA CORPORATION, 2012)

4 De uma maneira simplificada, pode-se dizer que thread é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas paralelamente.

2.6.2. CPU e GPU

Normalmente o primeiro passo no desenvolvimento de um programa é a preocupação de que o código fonte gerado produza os resultados pretendidos. A grande maioria dos programadores nesta fase inicial, não está muito preocupado se o seu código fonte está sendo executado da melhor maneira pelo hardware, muito pelo contrário, é aceitável que o mesmo demore algumas dezenas ou até mesmo centenas de minutos desde que, produza resultados satisfatórios. Entretanto, uma vez que esta etapa inicial é concluída e o código fonte testado e validado, os programadores normalmente entram em uma nova etapa do desenvolvimento, a otimização do código fonte.

Nesta etapa, os programadores buscam executarem seus códigos fontes da melhor forma possível em seu hardware (ou senão muito próximo disto) e para isto se utilizam de diversas técnicas de otimização, dentre elas o paralelismo. Geralmente, a otimização envolve transformações que geram códigos mais eficientes de serem processados pelo hardware. Estas otimizações podem ser feitas pelo próprio programador, ou pelo compilador da linguagem que tenta localizar no código fonte, trechos que podem ser otimizados e os modifica para que tal aconteça. Entretanto, códigos otimizados pelos compiladores tem suas limitações e na maioria das vezes, ainda podem ser mais otimizados de forma manual pelo programador.

Normalmente um programa é escrito de forma sequencial e executado dentro da Unidade de Processamento (CPU) também de forma sequencial. Desta forma, o tempo gasto para a execução do programa na grande maioria das vezes, é o somatório dos tempos gastos para que cada instrução dentro do processador seja executada. Uma forma de

otimizar a velocidade de execução deste programa é utilizar o conceito de processamento paralelo que consiste em dividir o programa em partes independentes e executar cada uma destas partes em diferentes unidades de processamento ao mesmo tempo.

Motivados pelo alto custo computacional de aplicações gráficas, principalmente pela indústria de jogos, surge na década de 80 os aceleradores gráficos não programáveis que, na próxima década, passa a ser possível graças ao advento da microeletrônica integrar todos os elementos essenciais dos aceleradores gráficos em um único chip. Em 2001, surge o primeiro acelerador gráfico programável com muitas limitações, entretanto, com uma expressiva largura de banda da memória dos chips em relação a CPU e como consequência do posterior desenvolvimento dos aceleradores gráficos, surge a computação nas unidades de processamento gráfico (GPUs) (GLASKOWSKY, 2009).

O desempenho das GPUs passou a superar em muito o desempenho de processamento das CPUs, e com isto os pesquisadores e programadores passaram a usar de forma não muito convencional as GPUs para propósitos gerais e não mais simplesmente para computação gráfica. Tendo um novo mercado pela frente, a empresa NVIDIA em 2007 resolve introduzir um modelo de arquitetura em seus aceleradores gráficos, denominado CUDA (Compute Unified Device Architecture). Esta arquitetura passa a incluir componentes estritamente projetados para computação em GPU, facilitando desta forma a programação hibrida das GPUs com as CPUs.

Com a arquitetura CUDA voltada para a programação de propósito geral e não mais somente para gráficos, surge a possibilidade da utilização de linguagens de programação voltadas para este tipo de arquitetura como por exemplo o CUDA C e o CUDA Fortran. Neste cenário, as GPUs originalmente voltadas para aplicações gráficas, passam a desempenhar um papel importante na computação de alto desempenho pois, já

possuem em seu interior, toda uma arquitetura voltada para o processamento paralelo, ganhando assim espaço no desenvolvimento de sistemas que demandam alto custo computacional.

2.6.3. Taxonomia de Flynn

Diversos tipos de computadores paralelos, já foram construídos até a presente data e diversos pesquisadores tentaram classifica-los em uma taxonomia. Existem diversos modelos de classificação, porém, embora não muito abrangente, o mais aceito é o modelo de taxonomia de Michael Flynn (FLYNN, 1972) que, propõe a classificação das arquiteturas de computadores, de acordo com as suas entradas e saídas baseando-se em dois conceitos: sequência de instruções e sequência de dados.

Flynn se refere a sequência de instruções ou fluxo de instruções como sendo um conjunto de instruções a serem executadas e sequência de dados ou fluxo de dados como sendo conjunto de dados. Desta forma o modelo de Flynn (Taxonomia de Flynn) classifica como o fluxo de dados e o fluxo de instruções são organizados dentro de um processamento. A taxonomia de Flynn fica dividida em quatro categorias: SISD, SIMD, MISD e MIMD conforme ilustrado na Figura 15 e exemplificado abaixo.

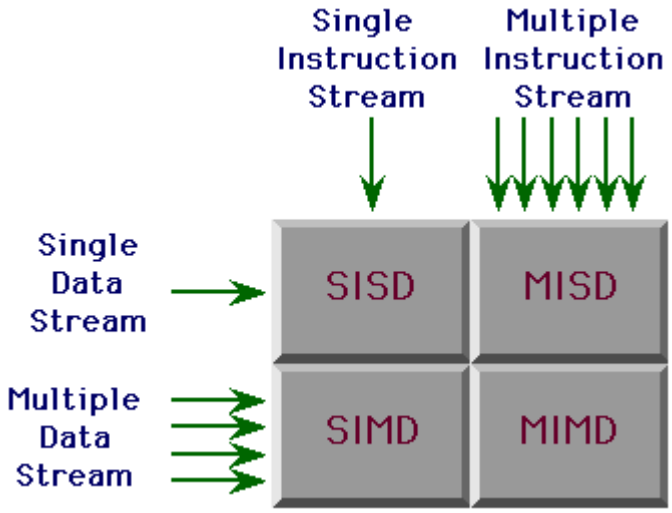

Figura 15 – Taxonomia de Flynn

Fonte: Adaptado de http://www-usr.inf.ufsm.br/~sandro/elc139/tarefa1.php, acessado em 21/02/2015

SISD – Single Instruction, Single Data – considera-se um único fluxo de instruções de entrada e um único fluxo de dados de entrada (Figura 16). É a clássica taxonomia sequencial que não explora o paralelismo, podendo compara-la a computadores de núcleo simples que possuem paralelismo de tarefas por time sharing (fatiamento de tempo do processador). Nesta classificação enquadram-se os computadores que sequem o padrão de Von Neumann, isto é, processadores seriais que executam uma instrução completa de cada vez, sequencialmente, cada uma delas manipulando um dado específico ou os dados daquela operação. Como exemplo, citamos o processamento escalar.

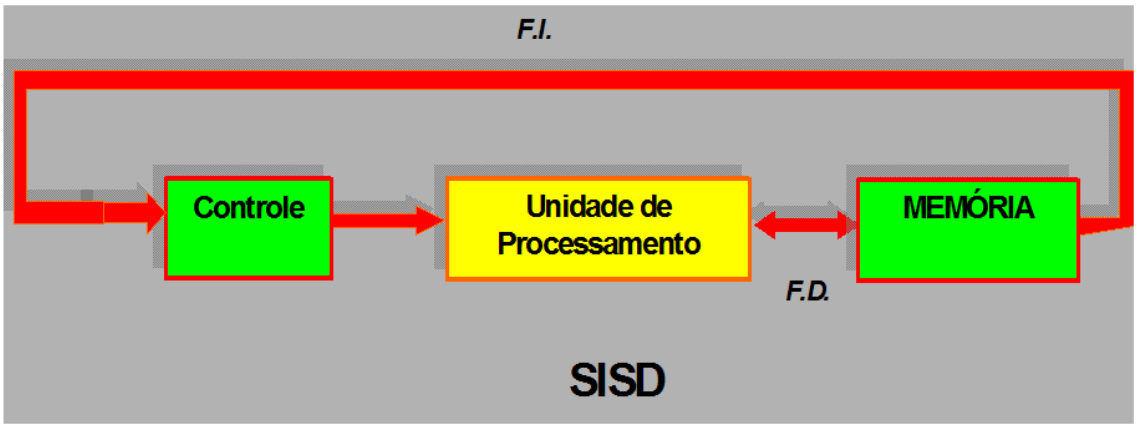

Figura 16 – Single Input Single Data
Fonte: Adaptado de http://coteia.icmc.usp.br, acessado em 21/02/2015

SIMD – Single Instruction, Multiple Data – considera-se um único fluxo de instrução de entrada e múltiplos fluxos de dados de entrada (Figura 17). Este sistema computacional processa múltiplos fluxos de dados simultaneamente a cada ciclo e executa operações, que sejam naturalmente paralelizáveis. Este tipo de arquitetura consiste em uma unidade de controle e N unidades de processamento, todo o processamento está sobre o controle de um único fluxo de instruções mas opera sobre N

fluxos de dados. Como exemplo citamos as GPUs que se encontram nesta categoria, bem como os grandes processadores vetoriais.

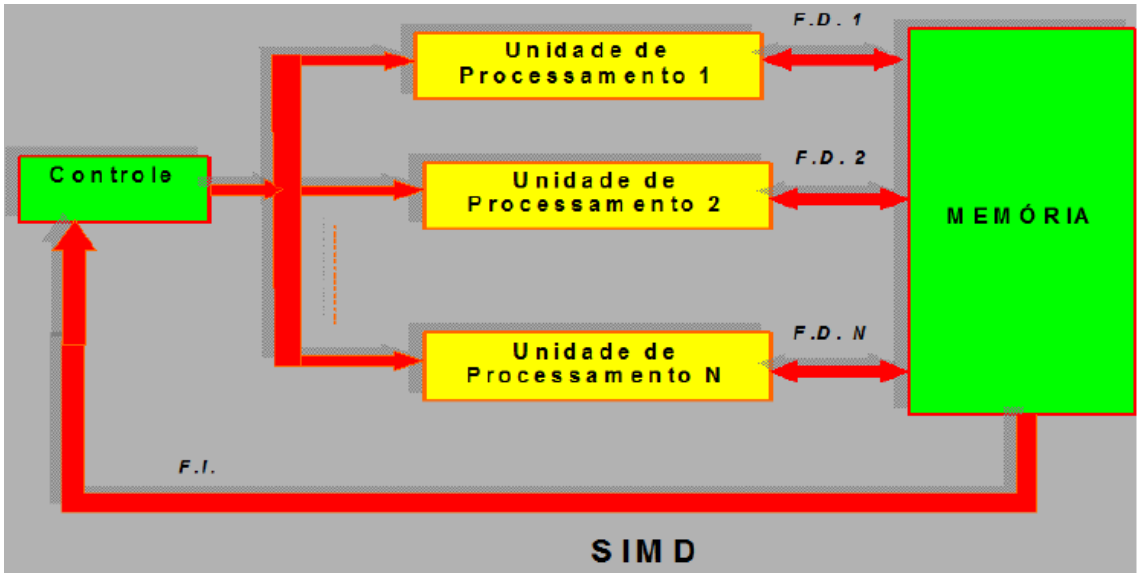

Figura 17 – Single Instruction Multiple Data
Fonte: Adaptado de http://coteia.icmc.usp.br, acessado em 21/02/2015

MISD – Multiple Instruction, Single Data – considera-se múltiplos fluxos de instruções de entrada e um único fluxo de dados de entrada (Figura 18). Este tipo de arquitetura pode usar múltiplas instruções para manipular um conjunto único de dados, como um vetor. N processadores cada um com sua unidade de controle e unidade de processamento, dividem uma mesma memória e executam diferentes instruções sobre o mesmo dado. É uma estrutura um pouco incomum, mas geralmente é utilizada por sistemas que requerem uma certa redundância a falhas. Um exemplo desta categoria de arquitetura é o do processador vetorial.

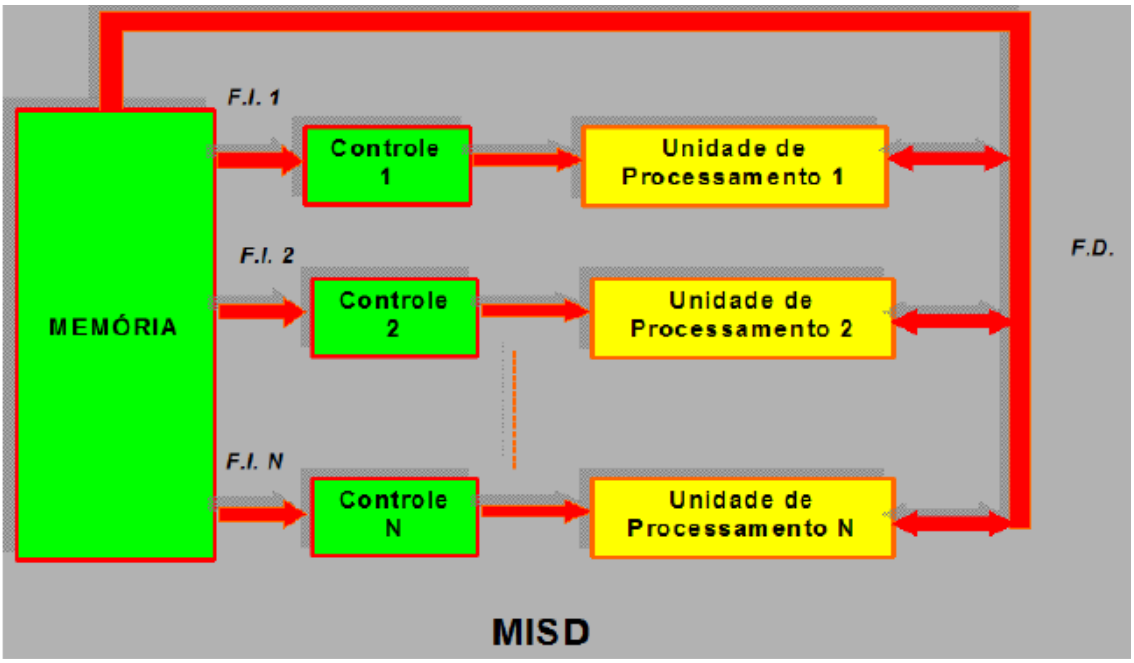

Figura 18 – Multiple Instruction Single Data
Fonte: Adaptado de http://coteia.icmc.usp.br, acessado em 21/02/2015

MIMD – Multiple Instruction, Multiple Data – considera-se múltiplos fluxos de instruções de entrada e múltiplos fluxos de dados de entrada (Figura 19). Pode ser considerada a categoria mais avançada tecnologicamente, produzindo elevado desempenho do sistema de computação. Consiste de N processadores distintos, controlados por N fluxos de instruções e operandos sobre N fluxos de dados. Os multiprocessadores e os multicomputadores encaixam-se nesta categoria, sendo que ambas as arquiteturas se baseiam em processadores múltiplos. Um exemplo desta categoria de arquitetura são as tecnologias de processamento distribuído tais como PVM e MPI.

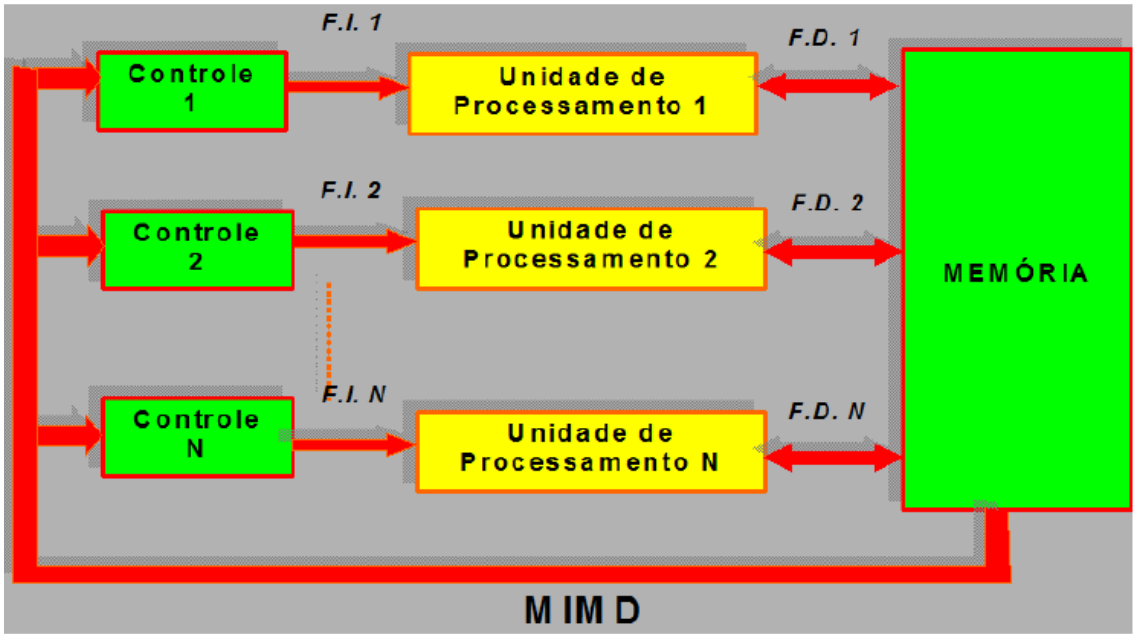

Figura 19 – Multiple Instruction Multiple Data
Fonte: Adaptado de http://coteia.icmc.usp.br, acessado em 21/02/2015

2.6.4. GPU – Graphic Processor Unit

De acordo com (HWU, KEUTZER e MATTSON, 2008), a indústria de semicondutores, vem seguindo por dois caminhos distintos no desenvolvimento de microprocessadores, o primeiro segue o caminho dos processadores de múltiplos núcleos (*multicore*) projetados para maximizar a velocidade de execução dos programas sequenciais, com a quantidade de núcleos dobrando a cada geração de semicondutores. Podemos citar como exemplo de processadores de múltiplos núcleos os processadores da Intel que, dependendo da versão, podem possuir mais de quatro núcleos por pastilha.

O segundo caminho segue o desenvolvimento de processadores com muitos núcleos (*manycore*), como as GPUs, com uma enorme quantidade de núcleos muito pequenos porem fortemente voltado para tarefas em paralelo (*multithread*). Podemos citar

como exemplos os processadores da NVIDIA que, dependendo da versão, podem apresentar mais de 5000 núcleos.

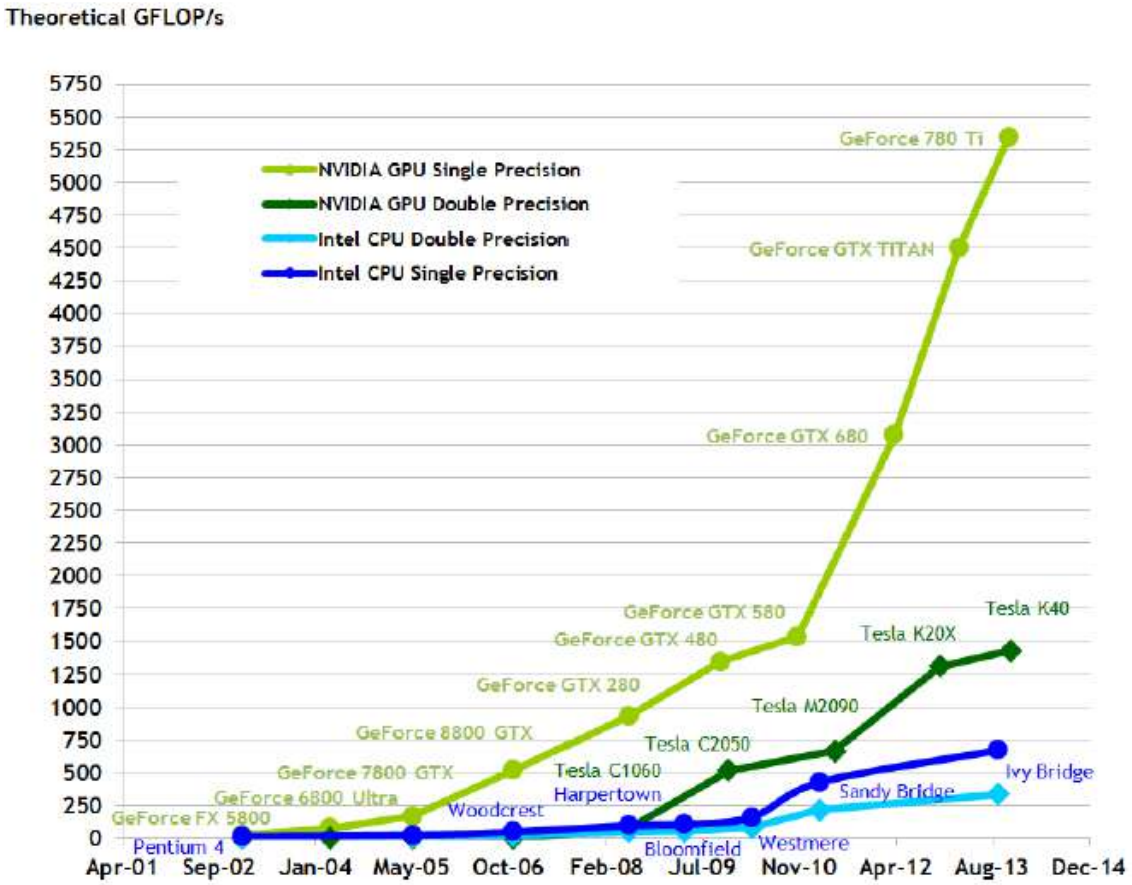

Figura 20 – Comparação entre CPU e GPU (Flops)
Fonte: (NVIDIA CORPORATION, 2014)

Impulsionado pela demanda crescente da computação de alto desempenho, as unidades com processadores gráficos programáveis ou simplesmente GPUs, tem evoluído para processadores *manycore* com uma alta capacidade de paralelismo e uma enorme potência computacional com altíssima largura de banda de memória. Podemos notar na Figura 20 uma enorme diferença entre a performance das GPUs em relação as CPUs, no que se diz respeito a quantidade de operações em ponto flutuante por segundo (Flops). Enquanto uma CPU Intel trabalhando com precisão simples consegue chegar próximo

aos 750 GFlops, uma GPU NVIDIA trabalhando com a mesma precisão consegue superar 5,25 TFlops, ou seja, uma performance de aproximadamente 7 vezes mais.

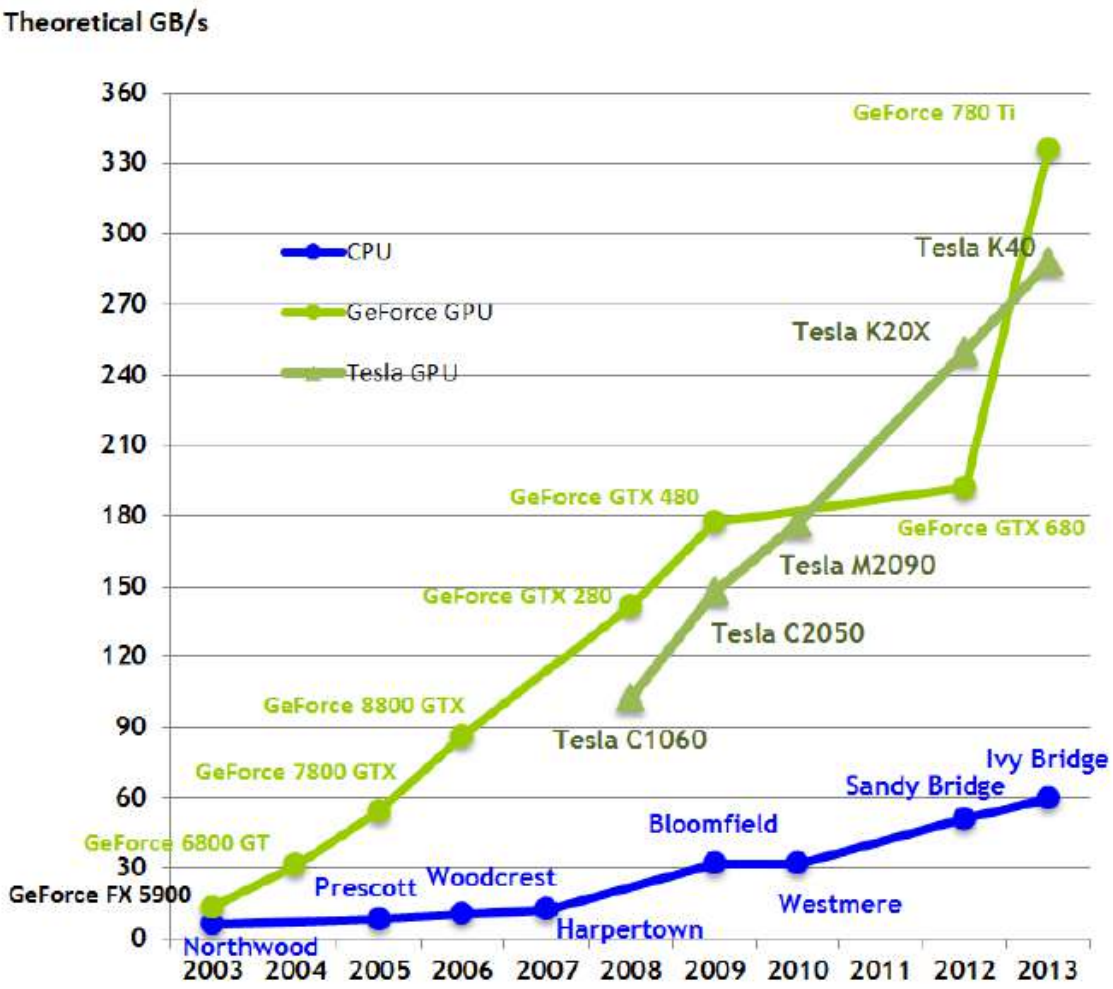

Figura 21 – Comparação entre CPU e GPU (Bytes/s)
Fonte: (NVIDIA CORPORATION, 2014)

Na Figura 21, temos o gráfico comparativo entre a largura de banda de memória entre uma CPU e uma GPU. Uma CPU Intel consegue transferir 60GB/s de dados, enquanto uma GPU NVIDIA transfere pouco mais que 330GB/s de dados, uma performance de aproximadamente 5,5 vezes maior. Uma das razões por existir esta discrepância entre a CPU e a GPU, é que a GPU é fabricada para o uso intensivo de computação paralela e portanto, concebida de tal forma que mais transistores são dedicados ao processamento de dados, em vez de cache de dados e controle de fluxo

existentes nas CPUs. Na Figura 22 podemos observar que a CPU com quatro unidades logicas aritméticas (ULAs) possui uma grande quantidade de memória cache, e faz uso de uma lógica de controle, para que as instruções em um único thread de controle sejam executadas fora de sua ordem sequencial nos diversos cores existentes na CPU, de forma paralela, reduzindo assim o tempo de acesso as instruções e dados de grandes e complexas aplicações.

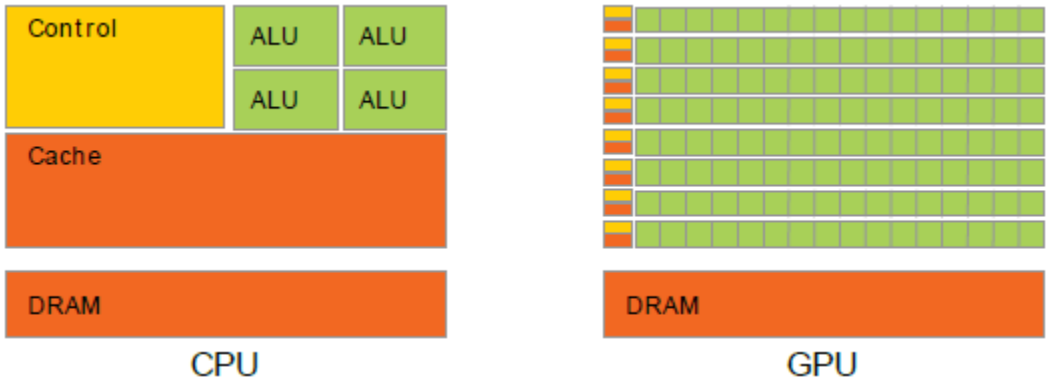

Figura 22 – Projetos diferentes entre CPUs e GPUs
Fonte: (NVIDIA CORPORATION, 2014)

A GPU representada na Figura 22, possui 128 ULAs e é especialmente desenvolvida para resolver os problemas que podem ser expressos como cálculos em forma paralela. O mesmo conjunto de instruções é executado nas diversas unidades lógicas aritméticas paralelamente e devido a este fato, existe uma menor necessidade de um controle de fluxo sofisticado e uma menor quantidade de memória cache.

Como originalmente a GPU foi projetada para trabalhar com imagens, enormes conjuntos de dados referentes aos pixels e vértices da imagem, são mapeados em forma de matrizes, para os threads paralelos. Entretanto, diversas aplicações que processam grandes quantidades de dados podem se beneficiar do modelo de programação paralela, bastando para isso que os dados sejam mapeados na forma matricial.

Diversos algoritmos que não pertencem ao campo da computação gráfica e processamento de imagem, podem ser acelerados por tratamentos de dados paralelo como por exemplo simulações nas áreas da engenharia nuclear e física aplicada. Segundo a taxonomia de Flynn, as GPUs são consideradas SIMD e por trabalharem principalmente com vetores também são conhecidas como processadores vetoriais.

As GPUs são tendência de baixo custo na computação paralela. Elas são poderosos coprocessadores, capazes de executar uma grande quantidade de instruções no mesmo ciclo de clock (em paralelo). Embora as GPUs tenham sido projetadas para melhorar o desempenho dos aplicativos de computação gráfica, seu uso foi estendido a muitos outros campos fora da computação gráfica.

Alguns usos de GPU para acelerar aplicações de engenharia nuclear podem ser vistos na literatura. (HEIMLICH, MOL e PEREIRA, 2011) propôs uma implementação baseada em GPU da solução baseada em diferenças finitas da equação de calor usando uma abordagem Red-Black Gauss-Seidel. (PEREIRA, MOL, *et al.*, 2013) usaram uma abordagem multi-GPU, para resolver uma simulação de Monte Carlo em física de reatores, neste trabalho, uma aceleração de mais de 2000 vezes foi alcançada usando 8 GPUs GTX-480. Mais recentemente, (HEIMLICH, SILVA e MARTINEZ, 2016) investigou uma implementação paralela com o uso de GPU para a queima em reatores do tipo PWR, onde se relacionam *speedups*[5] de aproximadamente 100 vezes. (MEDINA e

[5] Speedup é uma medida de desempenho do ganho em tempo, é usada para avaliar o fator de acelereção. Indica quantas vezes o algoritmo otimizado é mais rápido que sua versão original. É calculado pela razão do tempo do algoritmo original pelo tempo de sua versão otimizada.

CORTÉS, 2015) obtiveram *speedups* de cerca de 50 vezes em cálculos de campo de vento usando uma GTX-TITAN.

A GPU usada neste trabalho foi a GeForce GTX-680, a próxima geração de GPUs da NVIDIA habilitadas para CUDA. A GPU GTX-680 é baseada na arquitetura Kepler da NVIDIA e consiste de 4 GPC (Cluster de Processamento Gráfico) que compreendem 2 Streaming Multiprocessors (antigo SM e agora conhecido como SMX) com 192 núcleos CUDA cada, totalizando 1536 núcleos CUDA. A Figura 14 mostra um diagrama de blocos simplificado da GPU GeForce GTX 680, na Figura 23 podemos ver o diagrama de blocos simplificado do Cluster de Processamento Gráfico (GPC) com 2 SMXs (antigo SMs) e na Figura 24 um Streaming Multiprocessors (SMX).

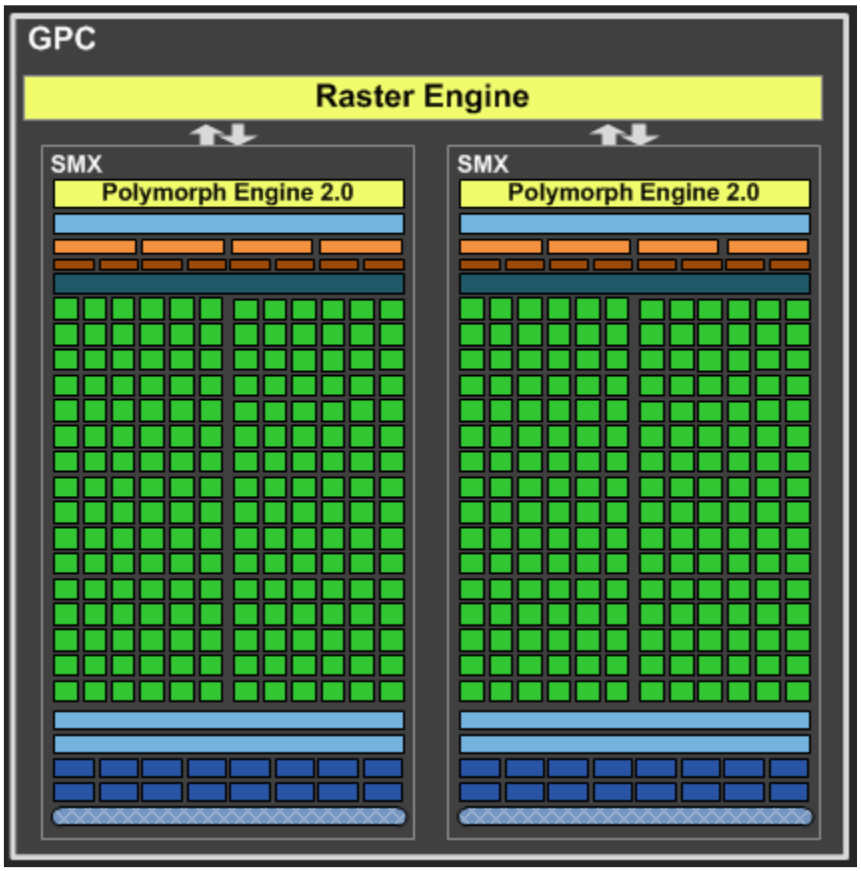

Figura 23 – Cluster de Processamento Gráfico (GPC) da NVIDIA GeForce GTX-680
Fonte: Adaptado de (NVIDIA CORPORATION, 2012)

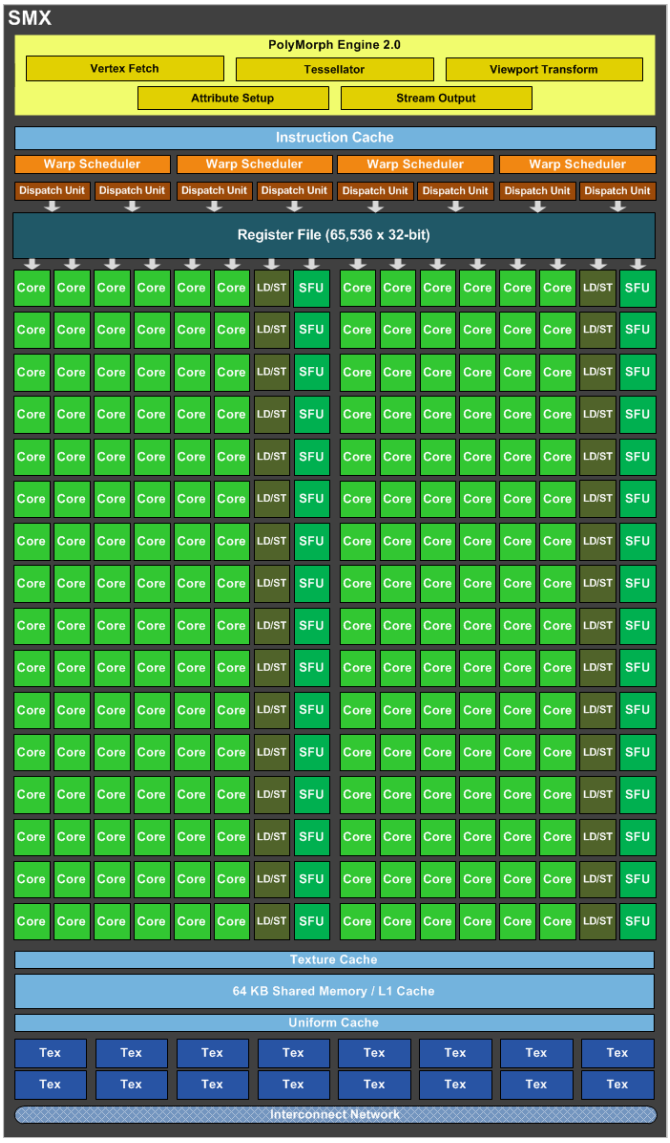

Figura 24 – Streaming Multiprocessors (SMX) da NVIDIA GeForce GTX 680
Fonte: (NVIDIA CORPORATION, 2012)

2.7. ARQUITETURA CUDA

A Compute Unified Device Architecture (CUDA) é um compilador baseado na linguagem C. O gerenciador global de blocos da GPU (GPU global block scheduler) gerencia o paralelismo de malha grossa no nível dos blocos de threads em todo o chip.

Quando uma função CUDA (*kernel*) é iniciada, informações para o grid (um grupo de blocos) são enviadas da CPU (*host*) para a GPU (*device*). A unidade de

distribuição de trabalho (*work distribution unit*) lê estas informações e envia dos blocos constituintes de threads para o SMX com capacidade disponível. A unidade de distribuição de trabalho (*work distribution unit*) envia blocos de threads (de uma maneira *round-robin*[6]) para os SMXs, que possuem recursos suficientes para executá-lo. O objetivo da unidade de distribuição de trabalho é distribuir de forma uniforme os threads em todos os SMXs para maximizar as oportunidades de execução paralelas. A Figura 25 ilustra a estrutura básica de um programa CUDA simples e sua comparação com a versão sequencial.

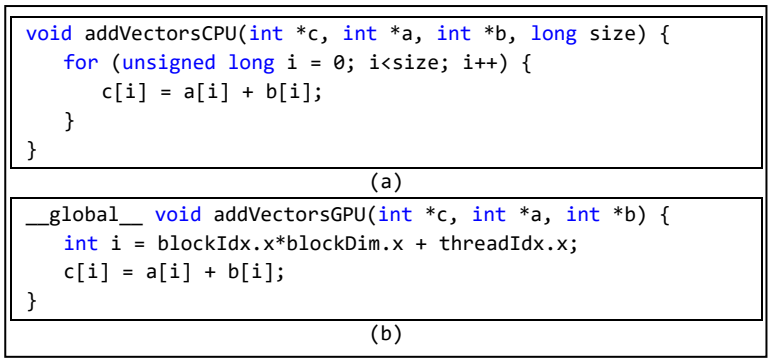

Figura 25 – Função simples para somar dois vetores (a) Sequencial e (b) código CUDA

Observe que, em vez do loop for (Figura 25a), que adiciona sequencialmente cada elemento do vetor, a função CUDA (Figura 25b), acrescenta apenas a posição i. A razão é que esta função, chamada CUDA kernel, é executada por um thread. Cada segmento executa a função do kernel para uma posição diferente no vetor, ao mesmo tempo, em

---

6 Round-robin (RR) é um dos algoritmos mais simples de agendamento de processos. Este algoritmo, atribui frações de tempo para cada processo em partes iguais (cada processo n recebe 1/n do tempo da CPU), de forma circular, manipulando todos os processos sem prioridades. Escalonamento do tipo Round-Robin é simples e fácil de implementar.

paralelo. No CUDA, os threads são organizados em blocos e podem ser identificados por seus blockId (índice do bloco) e threadId (índice do segmento dentro do bloco).

Na linguagem C, os parâmetros são transparentemente passados para a função usando a memória da CPU. Em CUDA, como as funções são executadas em GPU, os parâmetros devem ser copiados do Host (CPU) para a memória do Device (GPU). No final da execução, os resultados devem ser copiados de volta do Device para a memória do Host. O pedaço de código na Figura 26 mostra a alocação de memória no Host, a transferência de dados entre Host e Device, a chamada da função kernel e a transferência de dados entre o Device e o Host.

```
// Alocação da memória CUDA na GPU
cudaMalloc((void**)&d_c, size*sizeof(int));
cudaMalloc((void**)&d_a, size*sizeof(int));
cudaMalloc((void**)&d_b, size*sizeof(int));

// Copia a entrada de dados da memória do Host para a memória do Device
cudaMemcpy(d_a, a, size*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size*sizeof(int), cudaMemcpyHostToDevice);

// Chamada da função kernel na GPU
addVectorsGPU << <blocks, threadsPerBlock >> >(d_c, d_a, d_b);

// Copia a saída de dados da memória do Device para a memória do Host
cudaMemcpy(a, d_a, size*sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(b, d_b, size*sizeof(int), cudaMemcpyDeviceToHost);
```

Figura 26 – Alocação de memória, transferência de dados e chamada da função kernel

Deve-se ter em mente que as transferências de dados entre host e device consomem certo tempo, que é função da quantidade de dados a serem transferidos. Portanto, se o tempo de processamento for pequeno quando comparado com o tempo de transferência, a aceleração no paralelismo pode ser reduzida.

2.8. OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS – PSO

Cada vez mais os processadores atuais vêm aumentando sua performance computacional, entretanto, alguns problemas de engenharia como por exemplo, a recarga do combustível nuclear em reatores, ainda permanecem computacionalmente custosos e mesmo com os processadores atuais podem demorar dias, semanas ou até mesmo meses para serem processados. Metaheuristicas como o algoritmo genético (GOLDBERG e HOLLAND, 1988), colônia de formigas (DORIGO, 1992), colônia de abelhas (PHAM, GHANBARZADEH, *et al.*, 2006) e otimização por enxame de partículas (KENNEDY e EBERHART, 1995) entre outros são geralmente aplicados em tais problemas para encontrar soluções ótimas quando possível.

O método metaheurístico desenvolvido por (KENNEDY e EBERHART, 1995) conhecido como otimização por enxame de partículas (Particle Swarm Optimization – PSO), foi inspirado no comportamento de enxames de cardumes biológicos como aves e peixes e em aspectos de suas adaptações sociais. Foi baseado em um modelo simplificado da teoria da inteligência de enxames (*swarm intelligence* - (BENI e WANG, 1989)), no qual um indivíduo faz uso da sua própria experiência e da experiência coletiva para atingir um determinado objetivo. O PSO vem cada vez mais demonstrando uma boa alternativa na solução de problemas de otimização na área da engenharia nuclear conforme pode ser visto em (WAINTRAUB, SCHIRRU e PEREIRA, 2009), (MEDEIROS e SCHIRRU, 2008), (MENESES, MACHADO e SCHIRRU, 2008).

O PSO é um algoritmo iterativo onde é simulado um enxame de estruturas candidatas a solução potencial, denominada partícula, possuindo velocidade e valores de aptidão (fitness, a qual é avaliada pela função objetivo a ser otimizada) que estão procurando em um espaço n-dimensional (espaço de busca) por regiões que possuem um

alto valor de interesse (solução ótima) (Figura 27). Em resumo, cada partícula é tratada como um ponto em um espaço n-dimensional, ajustando sua trajetória baseada na própria experiência e na experiência coletiva. A cada iteração, cada partícula é atualizada por uma taxa de variação de posição (velocidade) em todas as n-dimensões, fazendo com que as partículas tendam gradualmente aos melhores valores históricos denominados pBest e gBest.

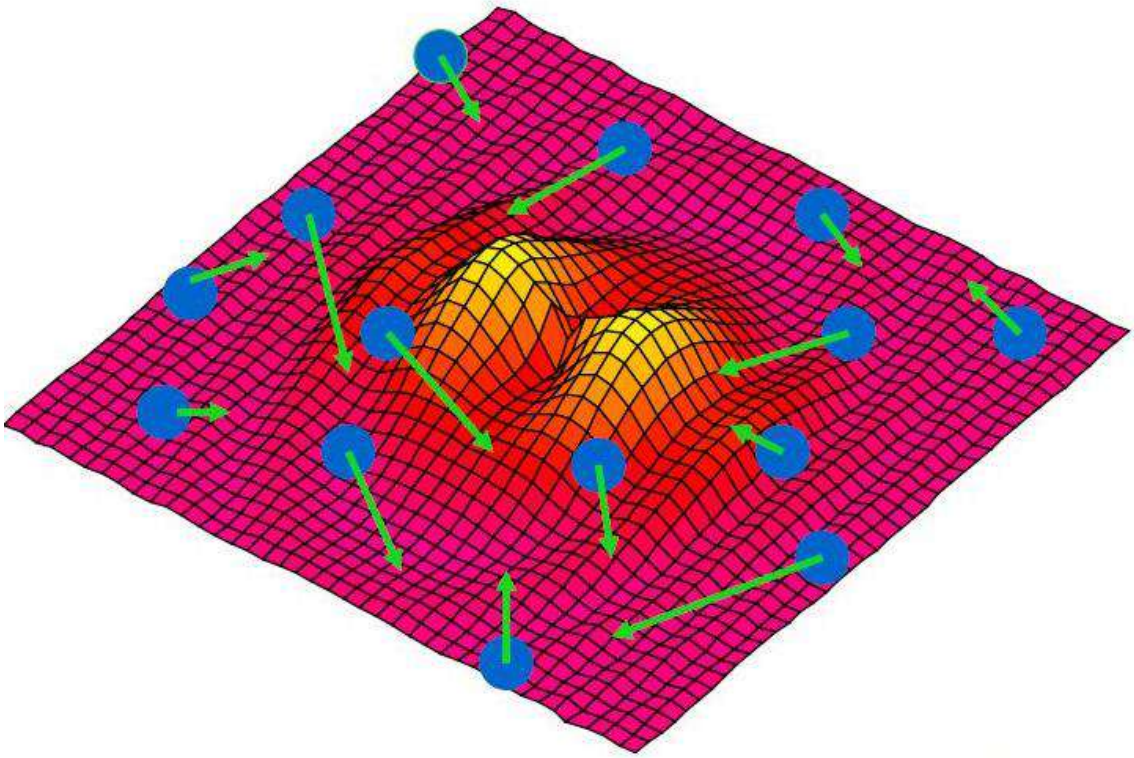

Figura 27 – Partículas observando um espaço de busca
Fonte: Adaptado de http://slideplayer.com.br/slide/1747559, acessado em 03/03/2017

Na Figura 28, podemos observar em um espaço de busca bidimensional a atualização do vetor posição de uma partícula. A mudança da posição da partícula e de sua velocidade, é influenciada pela sua posição do local de melhor fitness ou seja, guiada por sua própria experiência (pBest), mas também é guiada em direção das posições que

possuem a melhor fitness no espaço de pesquisa que são a melhor solução do grupo (gBest), desta forma, o enxame se desloca gradualmente em direção da melhor solução.

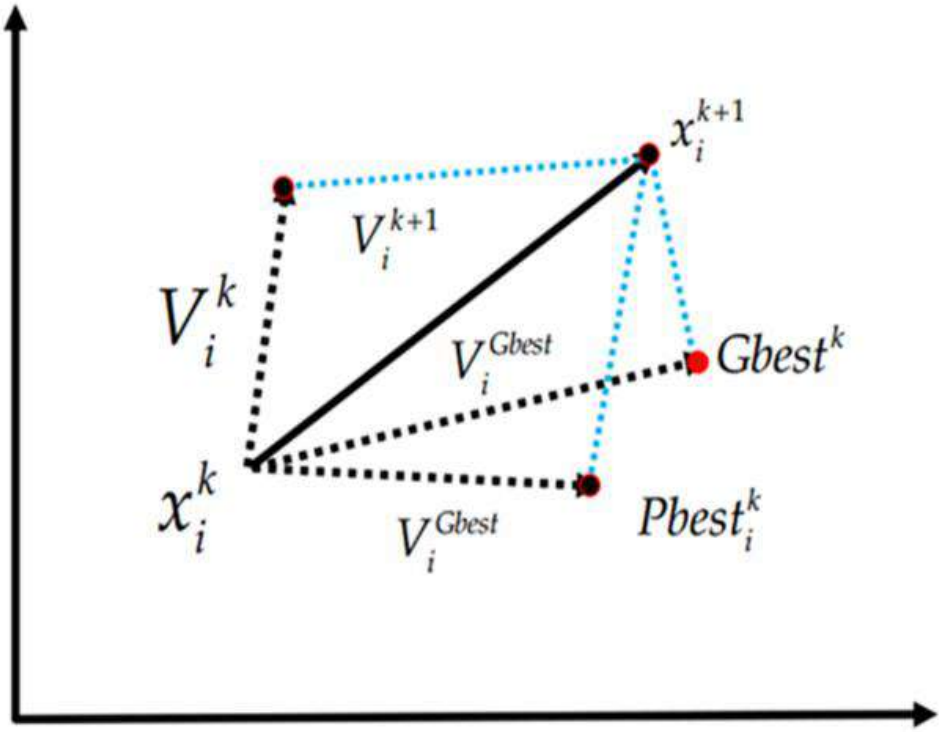

Figura 28 – Mecanismo de busca do PSO
Fonte: Adaptado de: http://pubs.sciepub.com/jit/2/1/3, acessado em 03/03/2017

As regras de atualização para a velocidade e posição das partículas do PSO podem ser descritas da seguinte forma: Sejam,

- $\overrightarrow{X_i}(t) = \{x_{i,1}(t), \cdots, x_{i,n}(t)\}$: a posição da partícula (i), candidata à solução do problema, na qual $x_{i,n}(t)$ representa uma variável da solução.
- $\overrightarrow{V_i}(t) = \{v_{i,1}(t), \cdots, v_{i,n}(t)\}$: a velocidade ou taxa de variação da partícula (i) no tempo t, em um espaço de busca n-dimensional.
- $\overrightarrow{pBest_i}(t) = \{pBest_{i,1}(t), \cdots, pBest_{i,n}(t)\}$: a melhor posição histórica já encontrada de cada partícula i até o tempo t.

- $\overrightarrow{gBest_i}(t) = \{gBest_{i,1}(t), \cdots, gBest_{i,n}(t)\}$: a melhor posição histórica já encontrada pelo enxame até o tempo t.

Assim, a equação da velocidade da partícula pode ser descrita pela Equação (24).

$$v_{i,n}(t+1) = w.v_{i,n}(t) + c_1.r_1.\Big(pBest_{i,n}(t) - x_{i,n}(t)\Big) + c_2.r_2.\Big(gBest_{i,n}(t) - x_{i,n}(t)\Big) \quad (24)$$

E a regra de atualização da partícula pode ser descrita pela Equação (25).

$$x_{i,n}(t+1) = x_{i,n}(t) + v_{i,n}(t+1) \quad (25)$$

Sendo,

- w: o peso inercial, que controla o impacto da velocidade previa na velocidade corrente;
- c_1: o coeficiente de aceleração da influência individual da partícula, ponderando a atração da partícula em direção a sua melhor posição histórica (pBest). Este fator indica a confiança que a partícula tem em si mesma;
- c_2: o coeficiente de aceleração da influência social da partícula, ponderando a atração da partícula em direção a melhor posição já ocupada pelo enxame (pBest). Este fator indica a confiança que a partícula tem no enxame;
- r_1 e r_2: números randômicos uniformemente distribuídos entre 0 e 1, gerados a cada iteração que, segundo (TRELEA, 2003), ajudam a diversificar a exploração do espaço de busca, ajudando o algoritmo a escapar mais facilmente de mínimos locais.

A Figura 29 exibe o algoritmo do PSO.

```
void Algoritmo_PSO (parâmetros){
   //Declarações e inicializações
   ...
   // O enxame é inicializado randomicamente
   for (i = 1; i <= n_particulas; i++){
      randomize (X[i]);
      randomize (V[i]);
   }
   for (Iter = 1; Iter <= Iter_Max; Iter++){
      for (i = 1; i <= n_particulas; i++){
         avalie (X[i]);
         atualiza (pBest[i], gBest);
         atualiza_velocidade (V[i]);
         atualiza_posicao (X[i]);
      }// i
   }// Iter
}
```

Figura 29 – Algoritmo do PSO

2.9. TÉCNICAS DE PARALELIZAÇÃO

2.9.1. Introdução

Em diversas áreas da engenharia e computação, cada vez mais vem surgindo problemas complexos que demandam alto poder de computação para serem resolvidos. A área nuclear não é uma exceção, e muitos destes problemas, envolvem cálculos complexos que mesmo nos computadores atuais demandam um certo tempo para serem executados. Uma forma de se reduzir o tempo de espera é, com a utilização de códigos paralelos e o uso de vários processadores. Diversas técnicas e bibliotecas disponíveis para a programação paralela CUDA já estão disponíveis, simplificando desta forma a paralelização de alguns problemas. Entretanto, dependendo do problema a ser otimizado, o uso de tais técnicas e bibliotecas nem sempre é possível.

Nesta seção, iremos apresentar algumas técnicas já existentes de paralelização utilizando a linguagem CUDA. De uma forma geral, o processo de paralelização de um

algoritmo, se inicia com a divisão do problema principal em diversos problemas menores. Em seguida, cada problema menor é tratado individualmente e distribuído nos diversos núcleos de uma GPU, exigindo do programador (entre outras ações como gerenciamento de memória, alocação de memória na GPU, etc.) garantir a independência dos dados, o sincronismo entre eles e se necessário sua comunicação pois cada dado, será executado em um SMX sem o conhecimento do outro (PARHAMI, 2002).

Em uma grande parte de algoritmos, diversos trechos do código possuem loops que demandam alto custo computacional, assim, a paralelização dos loops é uma técnica muito importante e segundo (HUANG e HSU, 2000) em seu trabalho "*A practical run-time technique for exploiting loop-level parallelism*", uma tarefa difícil. Loops apresentam um forte potencial para o paralelismo, entretanto existem diversas técnicas para fazê-lo, assim sendo, um dos obstáculos, é identificar qual a melhor técnica a ser utilizada para cada loop.

Uma das tarefas mais comuns na programação CUDA é o paralelismo de um loop utilizando um kernel. Existem diversas técnicas de reestruturação de loops como Loop Fusion, Loop Fission, Loop Reversal, Loop Interchanging, Loop Inversion, Loop Reversal, Loop Unrolling, Loop Skewing (WOLFE, 1996) e Grid-Stride Loop. Neste trabalho, iremos descrever algumas destas técnicas porem, a técnica Grid-Stride Loop será mais detalhada pois é amplamente utilizada nesta obra.

2.9.1. Loop Fusion

Loop fusion ou como também é conhecida, loop jamming, é uma técnica de transformação que substitui vários loops que interagem no mesmo intervalo em um único loop, desde que os dados manipulados pelos diversos loops, não façam referencias entre

si. Esta técnica nem sempre melhora a velocidade de execução pois, dependendo do tipo de arquitetura utilizada, cada instrução dentro do loop pode ser executada em threads diferentes. Um exemplo desta técnica pode ser visto na Figura 30.

```
// Algoritmo original
for (i = 0; i < 100; i++)
   a[i] = 1;
for (i = 0; i < 100; i++)
   b[i] = 2;

// Loop Fusion
for (i = 0; i < 100; i++){
   a[i] = 1;
   b[i] = 2;
}
```

Figura 30 – Loop Fusion

2.9.2. Loop Fission

Loop fission ou loop distribution, é uma otimização oposta ao loop fussion. Nesta otimização, o loop é quebrado em vários loops que possuem a mesma faixa de índice sendo que cada loop fica com uma parte do corpo do loop original. O objetivo desta técnica é dividir um loop grande em loops menores, de forma que em determinadas arquiteturas cada loop seja executado em um thread ou processador separado desta forma, aumentando a performance do algoritmo. Um exemplo da técnica loop fission pode ser visto na Figura 31.

```
// Algoritmo original
for (i = 0; i < 100; i++){
   a[i] = 1;
   b[i] = 2;
}

// Loop Fission
for (i = 0; i < 100; i++)
   a[i] = 1;
for (i = 0; i < 100; i++)
   b[i] = 2;
```

Figura 31 – Loop Fission

2.9.3. Loop Unrolling

Loop unrolling também conhecida como loop unwinding é uma técnica que tenta otimizar a velocidade de execução do programa em detrimento ao seu tamanho. Seu principal objetivo é acelerar o programa minimizando ou eliminando a aritmética de controle do loop, como por exemplo o controle dos ponteiros (do loop) e os testes de fim de loop. Como principal técnica, os loops podem ser reescritos como uma sequência repetida de instruções, deslocando-se o ponteiro para a próxima instrução sem passar pelo loop e, minimizando o número de repetições do loop em si. Na Figura 32 podemos ver um exemplo da técnica loop unrolling.

```
// Algoritmo original
for (i = 0; i < 100; i++){
    a[i] = 1;
}

// Loop Unrolling
for (i = 0; i < 100; i +=5){
    a[i] = 1;
    a[i + 1] = 1;
    a[i + 2] = 1;
    a[i + 3] = 1;
    a[i + 4] = 1;
}
```

Figura 32 – Loop Unrolling

2.9.4. Grid-Stride Loop

Vamos tomar como exemplo a Figura 33 que nos mostra uma implementação sequencial básica de um loop.

```
// Algoritmo original
void Grid_Stride_Loop(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

Figura 33 – Grid-Stride Loop – Algoritmo sequencial

Seguindo a orientação do guia de programação da própria Nvidia (NVIDIA CORPORATION, 2017), deveriam ser lançados threads para cada elemento de dado, o que na pratica significa paralelizar o loop acima. Supondo que temos threads suficientes para cobrir todo o tamanho do vetor, poderíamos reescrever o código em CUDA exibido na Figura 34.

```
// Algoritmo CUDA
__global__ void Grid_Stride_Loop (int n, float a, float *x, float *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a * x[i] + y[i];
}
```

Figura 34 – Grid-Stride Loop – Algoritmo CUDA

Esta forma de programação também é conhecida como kernel monolítico (*monolithic kernel*) pois assume uma enorme e única grade de threads para processar todo o vetor, bastando para isso, alocar a quantidade suficiente de threads na chamada da função kernel de forma que, cada thread se encarregue de uma posição do vetor. Tal técnica assume que a GPU possui uma quantidade de threads suficientes para cobrir todo o vetor, entretanto, quando o tamanho do vetor é maior que a quantidade de threads existentes na GPU, a técnica do kernel monolítico não mais pode ser usada pois, não conseguimos alocar na chamada da função kernel, todos os threads necessários para cobrir o vetor em uma única chamada. Ao invés de eliminar totalmente o loop ao paralelizar o código, podemos utilizar a técnica do grid-stride loop, conforme visto na Figura 35.

```
// Algoritmo Grid-Stride Loop
__global__ void Grid_Stride_Loop (int n, float a, float *x, float *y)
{
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n; i += blockDim.x * gridDim.x)
        y[i] = a * x[i] + y[i];
}
```

Figura 35 – Grid-Stride Loop – Algoritmo Grid-Stride Loop

Ao contrário de assumir que a grade de threads é grande o suficiente para alocar todo o vetor, esta técnica faz loops no vetor com um tamanho de grid de cada vez. Observe que o passo do loop é blockDim.x * gridDim.x que é o número total de segmentos do grid. Desta forma, se houver 2048 threads no grid, o thread 0 irá calcular os elementos 0, 2048, 4096, etc. Utilizando um loop com passo igual ao tamanho do grid, fica garantido que todo o endereçamento dentro dos wraps é uma unidade-passo, desta forma, obtemos o uso máximo dos threads assim como na versão monolítica. Quando o kernel é executado com um grid suficientemente grande para cobrir todas as iterações do loop, o grid-stride loop tem essencialmente o mesmo custo computacional da instrução if do kernel monolítico, pois o incremento do loop somente irá ser avaliado quando a condição do loop for verdadeira.

CAPITULO 3. IMPLEMENTAÇÃO PARALELA DO MODELO DE CAMPO DE VENTO NÃO DIVERGENTE

O programa do campo de vento baseado em GPU desenvolvido neste trabalho foi avaliado por meio de simulações considerando a região na vizinhança da central nuclear CNAAA, em Angra dos Reis, Brasil. O mapa mostrado na Figura 1 é exatamente a região coberta pelas simulações aqui descritas.

Nesta seção, são apresentados e discutidos os resultados qualitativos e quantitativos obtidos durante o desenvolvimento e aplicação da implementação do campo de vento baseado na GPU. Os principais pontos avaliados foram os seguintes:

I. Questões relativas ao refinamento dos critérios de convergência;
II. Validação da abordagem 3D-Red-Black;
III. Questões do refinamento do domínio computacional;
IV. Tempos de execução e aceleração obtidos;
V. Avaliação / análise qualitativa dos resultados considerando a literatura atual.

3.1. O ALGORITMO ORIGINAL

O objetivo principal deste trabalho é acelerar o algoritmo original, ou seja, o algoritmo que foi desenvolvido pelo PEN-COPPE/UFRJ, do programa west.furnas-1p, com pequenas modificações para reproduzir os mesmos resultados. Sendo assim, os valores obtidos pelo algoritmo original serão considerados como valores de referência e de uma maneira simplificada, no decorrer deste trabalho, utilizaremos o termo Original para cita-lo. Observe que o algoritmo de minimização de divergência original não trata da otimização de matriz (alocação), com o uso de técnicas de compressão de matriz ou

representações matriciais esparsas desta forma, a otimização da alocação de matriz está fora do escopo da presente investigação.

3.1.1. O Algoritmo Campo de Vento

De uma forma simplificada, o algoritmo do módulo do campo de vento não divergente sequencial utilizado no presente trabalho é o mostrado na Figura 36 e o tempo relativo ao processamento de cada função do algoritmo pode ser visto na Tabela 6.

```
Programa Campo_de_Vento
   Leitura_do_Terreno;
   Leitura_dos_Dados_Meteorologicos;
   West;
Fim Programa
```

Figura 36 – Algoritmo do módulo do Campo de Vento

Nome da função	Tempo de processamento relativo
Leitura_do_Terreno	0,149715 %
Leitura_dos_Dados_Meteorologicos	2,85E-4 %
West	99,85 %

Tabela 3 – Tempo de processamento relativo das funções do módulo sequencial do campo de vento

A função Leitura_do_Terreno é responsável pela leitura de um arquivo onde estão armazenadas as topografias do terreno. Este arquivo possui o formato de uma matriz de tamanho 67 x 43 onde cada célula possui um valor de 0 a 7 correspondentes aos 8 níveis de alturas variáveis conforme ilustrado na Tabela 2. Na Figura 37 tem-se o mapa da representação da topografia obtido do arquivo lido na função Leitura_do_Terreno. Para uma melhor visualização, foram suprimidos os valores de cota 0 (representados pelas células em branco). Na Figura 38 temos o mapa exibido por diferença de cores para cada

cota, neste mapa é possível visualizar de forma mais clara o relevo da região de interesse.

Esta função é responsável por 0,149715 % do tempo de execução total do programa.

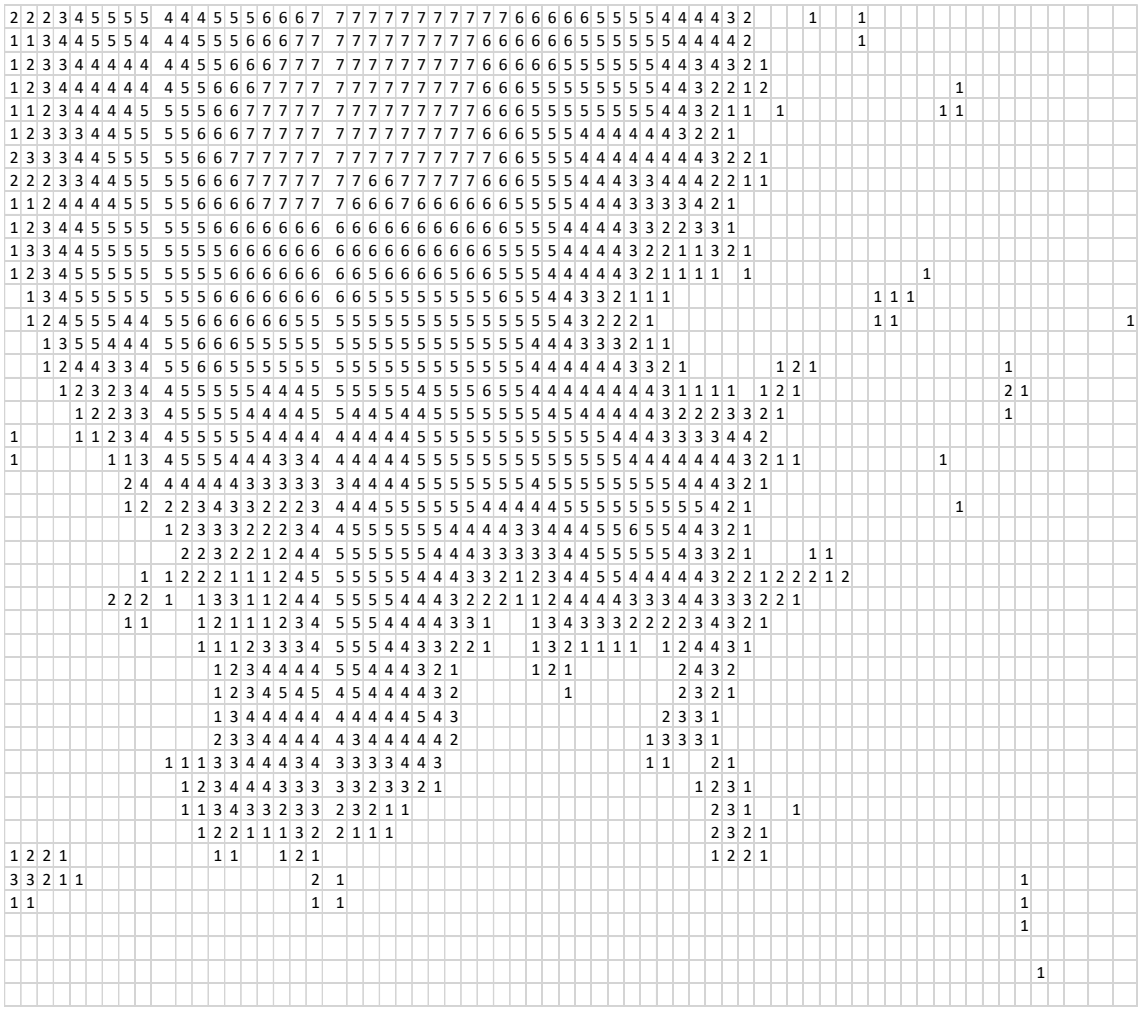

Figura 37 – Mapa da representação da topografia

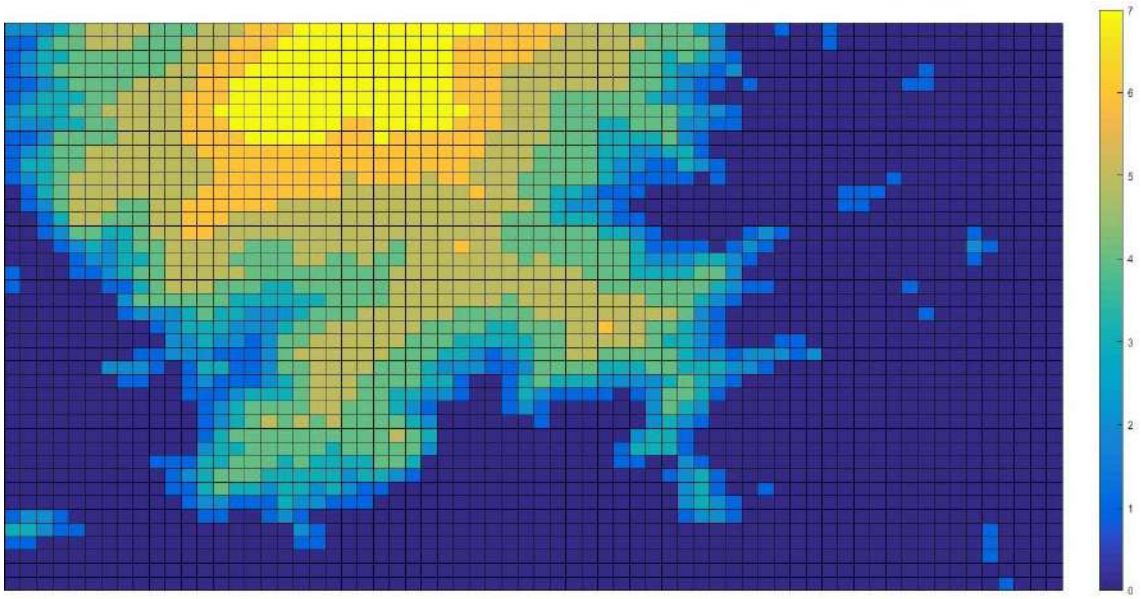

Figura 38 – Mapa da representação da topografia (cores)

A função Leitura_dos_Dados_Meteorologicos é responsável pela leitura de um arquivo onde estão armazenadas as informações adquiridas das estações meteorológicas situadas nas torres A, B, C e D. Um exemplo do conteúdo deste arquivo pode ser visto na Figura 39 onde pode ser observado que apenas a torre A possui 3 níveis de medição enquanto as demais torres possuem apenas um nível. As unidades dos valores contidos no arquivo, podem ser vistos da Tabela 9 até a Tabela 12.

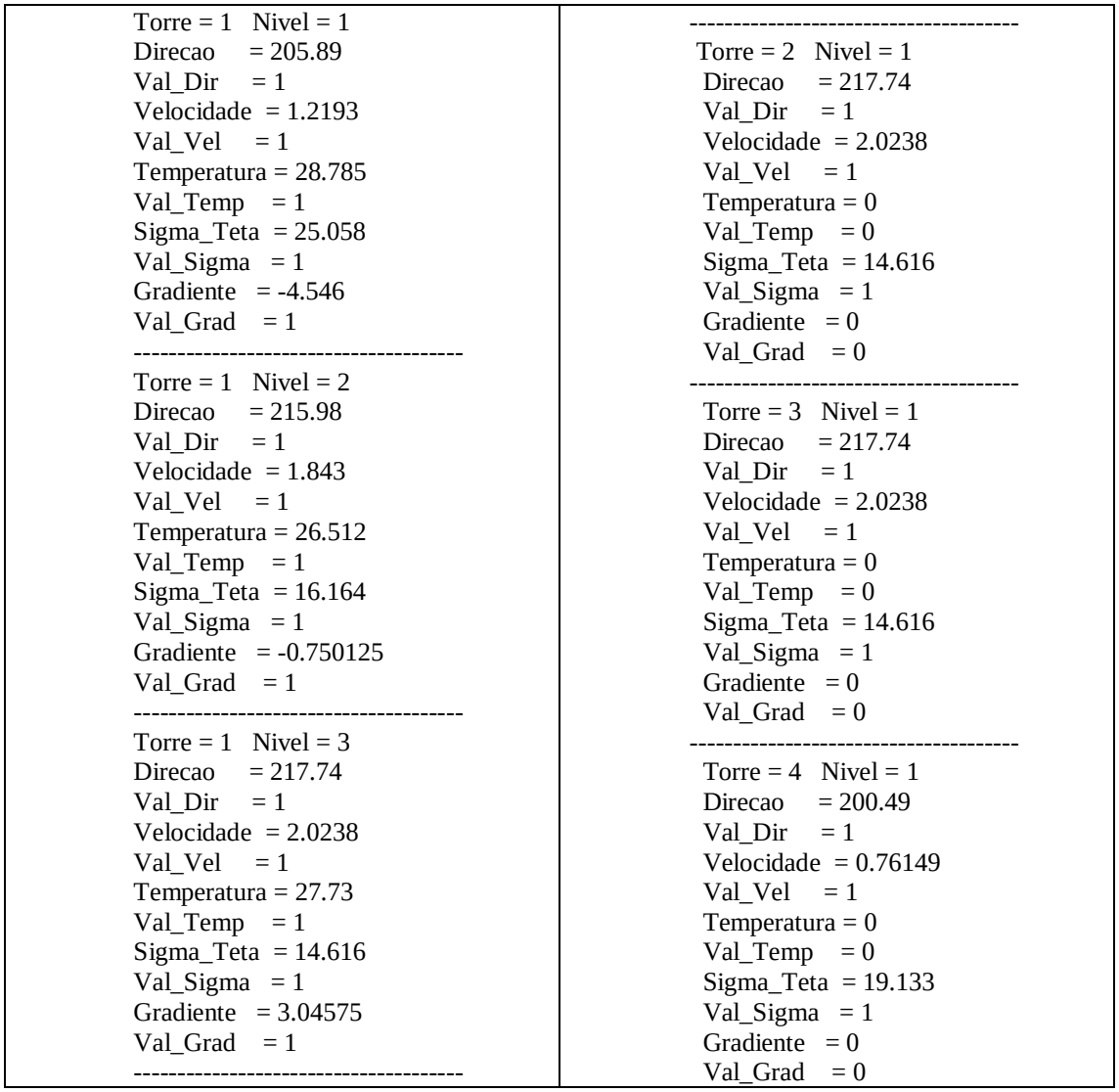

Figura 39 – Dados Meteorológicos

Ainda nesta função, é atribuído o grau de turbulência atmosférica, que pode ser caracterizado pelo valor do gradiente de temperatura entre dois níveis da atmosfera (Gradiente de Temperatura) no caso da torre A ou, pelo valor do desvio padrão das flutuações da direção horizontal da velocidade do vento (Sigma) nas demais torres conforme ilustrado pelo trecho do algoritmo na Figura 40.

```
void Leitura_dos_Dados_Meteorologicos (parâmetros){
   //Declarações e inicializações
   ...
   for (K = 1; K <= NUM_TORRES; K++){
      ...
      for (N = 1; N <= NIVEIS; N++){
         ...
         if (Torre[K].Val_Grad[N] == true)
            Estabilidade_Gradiente (parâmetros);
         else
            Estabilidade_Sigma (parâmetros);
      } // N
      ...
   } // K
   ...
}
```

Figura 40 – Atribuição do grau de turbulência atmosférica

O valor utilizado para designar a estabilidade e a relação entre a classe de estabilidade de Pasquill (PASQUILL, 1961) está exibido na Tabela 4, assim como a classe de estabilidade de Pasquill, a variação da temperatura do ar com a altura (Gradiente de Temperatura) e a distribuição dos valores do desvio padrão da direção horizontal do vento (Sigma) estão relacionados na Tabela 5. A função Leitura_dos_Dados_Meteorologicos é responsável por 2,85E-4 % do tempo total de processamento do programa.

Valor da estabilidade	Classe de estabilidade de Pasquill	Tipo de Vento
1,0	A	Fortemente Instável
2,0	B	Moderadamente Instável
3,0	C	Ligeiramente Instável
4,0	D	Neutra
5,0	E	Ligeiramente Estável
6,0	F	Moderadamente Estável
7,0	G	Fortemente Estável

Tabela 4 - Valores designados para a estabilidade

Valor da estabilidade	Gradiente de Temperatura (ºC/100m)	Sigma (Graus)
1,0	Gradiente ≤ -1,9	Sigma ≥ 22,5
2,0	-1,9 < Gradiente ≤ -1,7	22,5 > Sigma ≥ 17,5
3,0	-1,7 < Gradiente ≤ -1,5	17,5 > Sigma ≥ 12,5
4,0	-1,5 < Gradiente ≤ -0,5	12,5 > Sigma ≥ 7,7
5,0	-0,5 < Gradiente ≤ 1,5	7,7 > Sigma ≥ 3,8
6,0	1,5 < Gradiente ≤ 4,0	3,8 > Sigma ≥ 2,1
7,0	Gradiente > 4,0	Sigma < 2,1

Tabela 5 - Valores designados para o gradiente e sigma

A função West é a responsável por todo o cálculo do campo de vento não divergente e conforme ilustrado na Tabela 3, esta função ocupa 99,85 % de todo o tempo de processamento do programa. Uma vez que esta função representa a maior parte do tempo gasto, neste trabalho, esta função será investigada e otimizada.

3.1.2. O Algoritmo West

O Algoritmo da função West do módulo do campo de vento não divergente sequencial utilizado no presente trabalho é o mostrado na Figura 41 e o tempo relativo ao processamento de cada subfunção pode ser visto na Tabela 6. Tempos relativos a inicialização de vetores e demais códigos necessários ao funcionamento do programa e que não fazem parte do modelo de diagnóstico do campo de vento, possuindo tempo de execução insignificante, foram classificados como “Restante da Função West” na Tabela 6.

```
Função West
    Interpolacao_da_Estabilidade;
    Extrapolacao_Vertical;
    Inicializacao_do_Campo_de_Vento;
    Adicao_do_Terreno;
    Interpolacao_da_Velocidade;
    Velocidade_Zero;
    Calculo_da_Transparencia;
    Minimizacao_da_Divergencia;
    Remocao_da_Divergencia;
Fim Função West
```

Figura 41 – Algoritmo da função West

Nome da função	Tempo de processamento relativo
Interpolacao_da_Estabilidade	0,62%
Extrapolacao_Vertical	0,00%*
Inicializacao_do_Campo_de_Vento	0,16%
Adicao_do_Terreno	0,00%*
Interpolacao_da_Velocidade	5,95%
Velocidade_Zero	0,02%
Calculo_da_Transparencia	0,27%
Minimizacao_da_Divergencia	92,23%
Remocao_da_Divergencia	0,59%
Restante da Função West	0,01%* (Cada um < 0,002%)

* Tempo insignificante ($<10^{-3}$%)

Tabela 6 – Tempo de processamento relativo das subfunções da função West.

3.1.2.1. – Interpolação da Estabilidade

O algoritmo responsável pela interpolação da estabilidade pode ser visto na Figura 42 e ocupa 0,62% do tempo de processamento do programa.

O Campo de estabilidade é produzido realizando-se, em cada nível de altura, uma interpolação dos dados fornecidos e designando-se o valor zero as células obstáculo; a

interpolação é efetuada ponderando os dados com o inverso do quadrado da distância entre a célula de tomada e a célula de cálculo.

```
void Interpolacao_da_Estabilidade (parâmetros){
   //Declarações e inicializações
   ...
   for (K = NZ; K >= 1; K--){
      for (J = 1; J <= NY; J++){
         for (I = 1; I <= NX; I++){
            // Atribuição do valor zero para a estabilidade nas células obstáculo
            Estab[I][J][K] = 0.0;
            ...
            // Numero de torres de estabilidade válidas
            for (L = 1; L <= NumEsd; L++){
               ...
               // Soma do quadrado dos lados
               RFOR = (pow (((Ies[L] - I) * DX), 2) + pow (((Jes[L] - J) * DY), 2));
               ...
               // Somatório da estabilidade da torre pelo quadrado da distância
               Estab[I][J][K] = Estab[I][J][K] + Ess[L][K] / RFOR;
               // Somatório do inverso do quadrado da distância
               XNORM = XNORM + 1.0 / RFOR;
            }
            ...
            //Ponderação das células interpoladas
            if (XNORM > 0)
               Estab[I][J][K] = Estab[I][J][K] / XNORM;
            ...
         } // I
      } // J
   } // K
}
```

Figura 42 – Algoritmo da Interpolação da Estabilidade

3.1.2.2. – Extrapolação Vertical

O algoritmo responsável pela extrapolação vertical pode ser visto na Figura 43 e ocupa um tempo desprezível de processamento do programa. Nesta função, os valores observados nas torres para o modulo e a direção da velocidade são extrapolados ao longo da coordenada vertical, e as componentes horizontais da velocidade U e V são, então, determinadas nas células com velocidades medidas ou extrapoladas.

A extrapolação é efetuada, considerando que todas as células acima da última célula com velocidade ou direção medidas (torres), possuem o mesmo valor de velocidade

e direção que esta última. Nas velocidades extrapoladas é associado um fator de confiança igual a 10^{-3} e nas velocidades medidas um valor de 0,8.

```
void Extrapolacao_Vertical (parâmetros){
   //Declarações e inicializações
   ...
   // Numero de torres de estabilidade válidas
   for (L = 1; L <= NumEwd; L++){
      for (K = 1; K <= NZ; K++){
         // Atribui fator de confianca para velocidades medidas (0,8)
         Eww[L][K] = Eww[L][1];
         // Verifica se velocidade ou direção não foi medida
         if ((Ews[L][K] == 0) | (Ewd[L][K] == 0)){
            // Extrapola velocidade e direção
            Ews[L][K] = Ews[L][K - 1];
            Ewd[L][K] = Ewd[L][K - 1];
            // Atribui fator de confiança para velocidades extrapoladas (1,0E-3)
            Eww[L][K] = 1.0E-3;
         }
         // Determinacao das componentes horizontal U e V da velocidade das células
         WU[L][K] = - Ews[L][K] * sin (Ewd[L][K] * π / 180);
         WV[L][K] = - Ews[L][K] * cos (Ewd[L][K] * π / 180);
      } // K
   } // L
}
```

Figura 43 – Algoritmo da Extrapolação Vertical

3.1.2.3. – Inicialização do Campo de Vento

Esta função é responsável por 0,16% do tempo total de processamento e inicializa as componentes do vento (U, V e W) com valores nulos. Seu algoritmo pode ser visto na Figura 44.

```
void Inicializacao_do_Campo_de_Vento (parâmetros){
   //Declarações e inicializações
   ...
   for (K = 1; K <= NZ + 1; K++){
      for (J = 1; J <= NY + 1; J++){
         for (I = 1; I <= NX + 1; I++){
            if ((K != NZ + 1) && (I != NX + 1))
               Vento.V[I][J][K] = 0.0;
            if ((K != NZ + 1) && (J != NY + 1))
               Vento.U[I][J][K] = 0.0;
            if ((I != NX + 1) && (J != NY + 1))
               Vento.W[I][J][K] = 0.0;
         } // I
      } // J
   } // K
}
```

Figura 44 – Algoritmo da Inicialização do Campo de Vento

3.1.2.4. – Adição do Terreno

O algoritmo responsável pela adição do terreno pode ser visto na Figura 45 e ocupa um tempo desprezível do processamento do programa. Nesta função, a topografia da região nas torres é levada em consideração, deslocando as velocidades medidas e extrapoladas (nas torres) para os seus verdadeiros níveis K, e atribuindo a velocidade zero para as células obstáculo.

```
void Adicao_do_Terreno (parâmetros){
   //Declarações e inicializações
   ...
   // Numero de torres de estabilidade válidas
   for (L = 1; L <= NumEwd; L++){
      // Índices X e Y das torres
      I = Iew[L];
      J = Jew[L];
      // Topografia do terreno
      IC = Iht[I][J];
      ...
      NC = NZ - IC;
      if (NC != 0){
         for (K = 1; K <= NC; K++){
            ...
            // Deslocamento das velocidades
            WU[L][KK] = WU[L][KK - IC];
            WV[L][KK] = WV[L][KK - IC];
            Eww[L][KK] = Eww[L][KK - IC];
         }
      }
      // Atribuição de velocidade zero para as células obstáculo
      for (M = 1; M <= IC; M++){
         WU[L][M] = 0.0;
         WV[L][M] = 0.0;
         Eww[L][M] = 1.0E-6;
      }
   }
}
```

Figura 45 – Algoritmo da Adição do Terreno

3.1.2.5. – Interpolação da Velocidade

Um campo de velocidade é gerado fazendo-se uma interpolação das velocidades medidas e extrapoladas. Em cada plano horizontal, isto é, para cada nível K, a interpolação é efetuada para as componentes U e V de acordo com a seguinte ponderação vista na equação (26).

$$\begin{cases} U_{I,J,K} = \dfrac{\sum_{Torre=1}^{\text{Num.Torres}} U_{Torre,Nível} * \dfrac{Confiança_{Torre,Nível}}{Distância^2_{Torre \to I}}}{\sum_{Torre=1}^{\text{Num.Torres}} \dfrac{Confiança_{Torre,Nível}}{Distância^2_{Torre \to I}}} \\ V_{I,J,K} = \dfrac{\sum_{Torre=1}^{\text{Num.Torres}} V_{Torre,Nível} * \dfrac{Confiança_{Torre,Nível}}{Distância^2_{Torre \to J}}}{\sum_{Torre=1}^{\text{Num.Torres}} \dfrac{Confiança_{Torre,Nível}}{Distância^2_{Torre \to J}}} \end{cases}$$

(26)

Neste ponto, é importante observar que a componente da velocidade K (direção Z) é ainda nesta etapa nula em todo o domínio computacional. A função Interpolacao_da_velocidade é responsável por 5,95% do tempo de execução do programa e seu algoritmo pode ser visto na Figura 46.

```
void Interpolação_da_Velocidade (parâmetros){
   //Declarações e inicializações
   ...
   for (I = 1; I <= NX +1; I++){
      for (J = 1; J <= NY + 1; J++){
         for (K = 1; K <= NZ; K++){
            ...
            // Numero de torres de estabilidade válidas
            for (L = 1; L <= NumEwd; L++){
               // Cálculo do quadrado da distância na coordenada X e Y
               RSQDX = pow ((Iew[L] - I + 0.5), 2) * DX2 + pow ((Jew[L] - J), 2) * DY2;
               RSQDY = pow ((Iew[L] - I), 2) * DX2 + pow ((Jew[L] - J + 0.5), 2) * DY2;
               //Somatorio da componente da velocidade na torre L, fator de confiança e
                  inverso do quadrado da distância
               if (J != NY + 1)
                  Vento.U[I][J][K] = Vento.U[I][J][K] + WU[L][K] * Eww[L][K] / RSQDX;
               if (I != NX + 1)
                  Vento.V[I][J][K] = Vento.V[I][J][K] + WV[L][K] * Eww[L][K] / RSQDY;
               // Somatório do fator de confiança da torre L
               XNORMA = XNORMA + Eww[L][K] / RSQDX;
               YNORMA = YNORMA + Eww[L][K] / RSQDY;
            }
            // Média das NumEwd torres para cada célula
            if (J != NY + 1)
               Vento.U[I][J][K] = Vento.U[I][J][K] / XNORMA;
            if (I != NX + 1)
               Vento.V[I][J][K] = Vento.V[I][J][K] / YNORMA;
         } // K
      } // J
   } // I
}
```

Figura 46 – Algoritmo da Interpolação da Velocidade

3.1.2.6. – Velocidade Zero

Esta rotina, ocupa um tempo desprezível do processamento do programa e é responsável pela atribuição do valor zero as componentes de velocidade nas faces das células obstáculo. Seu algoritmo pode ser visto na Figura 47.

```
void Velocidade_Zero (parâmetros){
   //Declarações e inicializações
   ...
   for (I = 1; I <= NX; I++){
      for (J = 1; J <= NY; J++){
         ...
         // Atribui a altura do terreno
         K = Iht[I][J];
         if (K > 0)
            // Zera as faces das células obstáculo
            for (KK = 1; KK <= K; KK++){
               Vento.U[I][J][KK] = 0.0;
               Vento.V[I][J][KK] = 0.0;
               Vento.W[I][J][KK] = 0.0;

               Vento.U[I + 1][J][KK] = 0.0;
               Vento.V[I][J + 1][KK] = 0.0;
               Vento.W[I][J][KK + 1] = 0.0;
         } // KK
      } // J
   } // I
}
```

Figura 47 – Algoritmo da Velocidade Zero

3.1.2.7. – Cálculo da Transparência

Os coeficientes de transparência ou coeficientes de transmissão, são quantidades que, definidas nas faces das células, introduzem os efeitos da estabilidade atmosférica e da topografia no processo de ajuste. São calculados da conforme visto na equação (27).

$$\begin{cases} \tau_{X,I,J,K} = \eta_I * \tau_{H,m_{I+1}} + (1 - \eta_I) * \tau_{H,m_I} \\ \tau_{Y,I,J,K} = \eta_J * \tau_{H,m_{J+1}} + (1 - \eta_J) * \tau_{H,m_J} \\ \tau_{Z,I,J,K} = \eta_K * \tau_{V,m_{K+1}} + (1 - \eta_K) * \tau_{V,m_K} \end{cases} \quad (27)$$

Onde,

- η_I, η_J, η_K: São a parte decimal do valor atribuido a estabilidade de referência[7] da face perpendicular ao eixo (I, J ou K).
- m_I, m_J, m_K: São a parte inteira do valor atribuido a estabilidade de referência da face perpendicular ao eixo (I, J ou K).
- $\tau_{H,m}, \tau_{V,m}$: São os coeficientes de transmissão horizontal e vertical respectivamente, definidos em função da classe de estabilidade de Pasquill, de acordo com a Tabela 7 (FABRICK, SKLAREW e WILSON, 1977).

Classe de Pasquill	A	B	C	D	E	F	G
τ_H	1	1	1	1	200	500	1000
τ_V	1,6	1,4	1,2	1	0,8	0,6	0,4

Tabela 7 – Coeficientes de transmissão horizontal e vertical

Os coeficientes de transmissao nas faces das células obstaculo são nulos. O algoritmo responsavel pelo calculo da transparencia pode ser visto na Figura 48 e é responsável por 0,27% do tempo de processamento do programa.

[7] Se no interior do domínio, é a média aritmética das estabilidades das células em que a face é interface. Se na superfície do contorno do domínio, é o valor da estabilidade da célula a que pertence a face considerada.

```
void Calculo_da_Transparência (parâmetros){
   //Declarações e inicializações
   ...
   for (I = 1; I <= NX; I++){
      for (J = 1; J <= NY; J++){
         for (K = 1; K <= NZ; K++){
            ...
            // cálculo do coeficiente de transparência para a direção X
            ...
            // Verifica se é célula obstáculo
            if (Iht[I][J] < K){
               // Calcula o valor do coeficiente de transparência
               T.X[I][J][K] = Calc(Estab[I][J][K], TH);
            }
            Else {
               // Valor zero para o coeficiente de transparência das células obstáculo
               T.X[I][J][K] = 0.0;
            }
            ...
            // cálculo do coeficiente de transparência para a direção Y
            ...
            // cálculo do coeficiente de transparência para a direção Z
            ...
         } // K
      } // J
   } // I
}
```

Figura 48 – Algoritmo do Cálculo da Transparência

3.1.2.8. – Minimização da Divergência

O campo de velocidade gerado no algoritmo Interpolacao_da_Velocidade (Figura 46) é ajustado através de um processo iterativo, de forma que seja satisfeita em cada célula do reticulado, a condição de não divergência. Ao final de cada iteração, os valores das componentes U e V são corrigidos e incorporados ao processo iterativo através da média ponderada vista na equação (28).

$$\begin{cases} U_{I,J,K}^{corrigido} = \dfrac{U_{I,J,K}^{ajustado} + \sqrt{Confiança_{Torre,Nível}} * U_{Torre,Nível}}{1 + \sqrt{Confiança_{Torre,Nível}}} \\ V_{I,J,K}^{corrigido} = \dfrac{V_{I,J,K}^{ajustado} + \sqrt{Confiança_{Torre,Nível}} * U_{Torre,Nível}}{1 + \sqrt{Confiança_{Torre,Nível}}} \end{cases} \quad (28)$$

O referencial teórico para o ajuste das componentes de velocidade, já foi discutido todavia, as equações responsáveis pelo ajuste das componentes, são referenciadas na Figura 49, que exibe o algoritmo da Minimizacao_da_Divergencia que é responsável por 92,23% do tempo de processamento do programa. Neste trabalho, esta sub-rotina será o principal foco no processo de minimização e otimização pois, é a responsável por quase todo o tempo de processamento do programa campo de vento.

```
void Minimizacao_da_Divergencia (parâmetros){
 //Declarações e inicializações
 ...
 while (!critério parada) {
   for (K = 1; K <= NZ; K++){
    for (J = 1; J <= NY; J++){
     for (I = 1; I <= NX; I++){
     // Coeficiente total de transissão
     TOT = 1.0 / ((T.X[I+1][J][K] + T.X[I][J][K]) / DX2 +
                  (T.Y[I][J+1][K] + T.Y[I][J][K]) / DY2 +
                  (T.Z[I][J][K+1] + T.Z[I][J][K]) / DZ2 );
     // Divergente
     D = (Vento.U[I+1][J][K] - Vento.U[I][J][K]) / DX +
         (Vento.V[I][J+1][K] - Vento.V[I][J][K]) / DY +
         (Vento.W[I][J][K+1] - Vento.W[I][J][K]) / DZ ;                          // Eq.(4)
     // Diferença da velocidade de perturbação
     delta_Phi = 1.25 * TOT * D;   // 1.25 = Fator de relaxação                 // Eq.(17)
     // Correção da velocidade
     Vento.U[I][J][K]   = Vento.U[I][J][K]   + (delta_Phi*T.X[I][J][K])   / DX; // Eq.(18)
     Vento.U[I+1][J][K] = Vento.U[I+1][J][K] - (delta_Phi*T.X[I+1][J][K]) / DX; // Eq.(19)

     Vento.V[I][J][K]   = Vento.V[I][J][K]   + (delta_Phi*T.Y[I][J][K])   / DY; // Eq.(20)
     Vento.V[I][J+1][K] = Vento.V[I][J+1][K] - (delta_Phi*T.Y[I][J+1][K]) / DY; // Eq.(21)

     Vento.W[I][J][K]   = Vento.W[I][J][K]   + (delta_Phi*T.Z[I][J][K])   / DZ; // Eq.(22)
     Vento.W[I][J][K+1] = Vento.W[I][J][K+1] - (delta_Phi*T.Z[I][J][K+1]) / DZ; // Eq.(23)
     } // I
    } // J
   } // K
   // Correção das componentes U e V
   // Checa critério de convergência
   ...
 } // While
}
```

Figura 49 – Algoritmo da Minimização da Divergência

3.1.2.9. – Remoção da Divergência

Esta rotina é responsável por 0,59% do tempo de processamento do programa e, ao final da rotina Minimizacao_da_Divergencia, todas as divergências residuais são deslocadas para o topo do reticulado, de forma contínua para uma mesma prumada, adicionando os resíduos as componentes verticais das velocidades das células. Seu algoritmo pode ser visto na Figura 50.

```
void Remoção_da_Divergência (parâmetros){
   //Declarações e inicializações
   ...
   for (I = 1; I <= NX; I++){
      for (J = 1; J <= NY; J++){
         for (K = 1; K <= NZ; K++){
            Vento.W[I][J][K+1] = (Vento.U[I][J][K] - Vento.U[I+1][J][K]) * DZ / DX +
                                 (Vento.V[I][J][K] - Vento.V[I][J+1][K]) * DZ / DY +
                                  Vento.W[I][J][K];
         } // K
      } // J
   } // I
}
```

Figura 50 – Algoritmo da Remoção da Divergência

3.2. DADOS METEOROLÓGICOS DE ENTRADA

Neste trabalho, foi considerado um conjunto de 20 tomadas meteorológicas reais de ventos, adquiridos em estações situadas nas torres A, B, C e D, conforme ilustrado na Figura 1. Os diversos sensores situados nestas torres, possuem uma precisão de ±10% e as tomadas meteorológicas de temperatura e gradiente de temperatura, apenas são adquiridas na torre A pois somente esta, possui os sensores necessários para este tipo de aquisição. Conforme visto na Figura 39, esta torre possui três níveis de altura, entretanto, a seguir serão mostrados como torre A a média destes valores.

A fim de proporcionar uma maior diversidade sobre os padrões de vento, foram escolhidas aquisições de dados de diferentes épocas (meses) e horários do dia (Tabela 8).

Como resultado, foram obtidos ventos com uma ampla gama de direções (Tabela 9), velocidades (Tabela 10), Variações de velocidade (Tabela 11), Temperatura (Tabela 12), gradiente de temperatura (Tabela 12) e classes de estabilidade (Tabela 13).

Datas das aquisições		
Vento	**Dia**	**Hora**
#1	01/01/2015	00:15
#2	20/01/2015	16:00
#3	29/01/2015	21:00
#4	01/02/2015	12:00
#5	25/02/2015	09:00
#6	10/04/2015	22:30
#7	15/04/2015	03:45
#8	05/05/2015	07:15
#9	21/06/2015	05:45
#10	22/06/2015	11:45
#11	27/07/2015	09:30
#12	27/07/2015	15:30
#13	05/08/2015	20:00
#14	09/09/2015	03:15
#15	09/09/2015	14:15
#16	18/10/2015	05:00
#17	15/11/2015	22:00
#18	17/11/2015	18:00
#19	25/12/2015	00:15
#20	31/12/2015	23:45

Tabela 8 – Datas das aquisições dos ventos utilizados neste trabalho

Direção do vento (graus)				
Vento	**Torre A**	**Torre B**	**Torre C**	**Torre D**
#1	104,7467	74,2700	128,5500	128,5500
#2	6,0300	353,5200	325,7000	325,7000
#3	40,4920	138,8400	40,4100	40,4100
#4	33,2033	37,7400	37,7400	20,4900
#5	266,0963	281,3100	38,3800	282,7100
#6	268,3960	257,1240	258,4750	296,8500
#7	343,2610	179,3900	175,8900	276,7990
#8	309,0353	255,1930	150,2500	100,8600
#9	3,5580	169,2700	310,6300	222,4840
#10	51,7067	33,4600	315,3600	229,8510
#11	348,8120	48,9100	322,6500	336,6300
#12	338,8947	324,2500	299,3600	338,1400
#13	186,1847	169,1700	283,3300	224,4790

#14	206,9962	233,4410	191,6460	191,6460
#15	249,0803	264,9420	279,5170	279,5170
#16	30,8267	355,6700	310,1300	72,4000
#17	98,2000	81,7900	104,1700	101,8900
#18	63,3533	318,4000	295,9000	329,4500
#19	211,4459	128,2200	270,8800	289,7700
#20	245,0255	267,0180	273,7410	269,6200

Tabela 9 – Direções dos ventos (graus) utilizados neste trabalho

Velocidade do vento (m/s)				
Vento	**Torre A**	**Torre B**	**Torre C**	**Torre D**
#1	1,2309	1,5475	1,4950	1,4950
#2	3,0280	3,3277	3,7760	3,7760
#3	1,3072	1,2601	0,3376	0,3376
#4	1,6954	2,0238	2,0238	0,7615
#5	2,1178	3,3511	1,6609	1,5393
#6	3,2965	5,3883	2,4569	1,2641
#7	1,0955	1,1273	1,6879	0,8750
#8	1,1387	1,2606	2,9788	1,6979
#9	0,9782	1,8496	0,7207	1,6988
#10	0,9565	1,3627	1,6493	1,0254
#11	1,3115	1,1726	1,0355	0,1272
#12	2,5651	5,0357	4,0978	0,8127
#13	0,7009	1,1972	1,2312	0,9751
#14	1,5563	1,3561	2,0014	2,0014
#15	2,3821	2,5674	2,9597	2,9597
#16	2,6828	4,3041	2,5880	1,1361
#17	1,5295	1,9656	1,6317	1,4005
#18	1,3601	5,0449	3,5151	0,7248
#19	0,7638	1,0869	1,2903	0,5631
#20	1,0938	0,8259	1,4776	1,8600

Tabela 10 – Velocidades dos ventos (m/s) utilizados neste trabalho

Variação de Velocidade (graus)				
Vento	**Torre A**	**Torre B**	**Torre C**	**Torre D**
#1	18,0893	12,4440	11,3560	11,3560
#2	20,4540	19,1480	16,7830	16,7830
#3	23,2317	22,7600	17,0420	17,0420
#4	18,6127	14,6160	14,6160	19,1330
#5	39,9277	11,6070	39,8010	44,3830
#6	19,4756	10,6720	24,3650	75,6310
#7	46,7552	5,7756	12,6380	8,7964

#8	55,6367	64,3710	12,2840	14,6660
#9	18,4654	10,6840	77,4890	13,3690
#10	35,2140	36,5790	59,7270	24,0970
#11	11,9202	15,0080	25,1540	17,5280
#12	18,7008	5,8577	12,0100	42,0610
#13	1,3551	8,2421	10,8190	20,0240
#14	34,6080	30,9090	42,0140	42,0140
#15	24,3510	22,0780	19,0580	19,0580
#16	14,6793	6,2844	19,0990	55,7410
#17	22,5162	20,3310	24,2160	9,9333
#18	49,4330	6,9126	12,7870	63,1710
#19	18,9904	80,6420	10,7570	5,2208
#20	11,5884	47,3280	6,1925	13,6790

Tabela 11 – Variação das velocidades (graus) dos ventos utilizados neste trabalho

Torre A		
Vento	**Temperatura (ºC)**	**Gradiente de temperatura (ºC/100m)**
#1	28,0733	1,9662
#2	29,7140	-2,0994
#3	29,0817	0,8350
#4	27,6757	-0,7501
#5	28,7263	-1,4823
#6	23,6617	-0,3777
#7	23,2573	0,5009
#8	21,8757	0,9264
#9	17,9587	1,7408
#10	23,2833	-0,2485
#11	20,5963	-1,3728
#12	20,7637	-1,3201
#13	21,0540	0,2936
#14	23,0737	2,4511
#15	21,1933	-1,4004
#16	20,3837	-0,6964
#17	22,4163	-1,4380
#18	24,4820	-1,5638
#19	27,5350	0,8227
#20	28,9887	1,2976

Tabela 12 – Temperatura (ºC) e Gradiente de temperatura (ºC/100m) dos ventos utilizados neste trabalho

Classe de estabilidade (Pasquill)				
Vento	**Torre A**	**Torre B**	**Torre C**	**Torre D**
#1	F	D	D	D
#2	A	B	C	C
#3	E	A	C	C
#4	D	C	C	B
#5	C	D	A	A
#6	E	D	A	A
#7	E	E	C	D
#8	E	A	D	C
#9	F	D	A	C
#10	E	A	A	A
#11	D	C	A	B
#12	D	E	D	A
#13	E	D	D	B
#14	F	A	A	A
#15	D	B	B	B
#16	D	E	B	A
#17	D	B	A	D
#18	D	E	C	A
#19	E	A	D	E
#20	F	A	E	C

Tabela 13 – Classe de estabilidade (Pasquill) dos ventos utilizados neste trabalho

3.3. REFINAMENTO DO CRITÉRIO DE CONVERGÊNCIA

No programa original, devido a limitações computacionais da época em que foi desenvolvido, o critério de parada do algoritmo de minimização da divergência (Figura 49) foi um número fixo de 56 iterações. Observou-se, porém, que esta abordagem não é efetiva para atingir valores de divergência adequados.

Simulações foram feitas com o algoritmo original considerando $D \leq 10^{-3}$ (onde D é a velocidade divergente) como critério de parada para os 20 ventos reais observados e seus resultados exibidos na Tabela 14. A coluna Iterações representa a quantidade de iterações necessárias para atingir o critério de parada, a coluna Célula mostra as

coordenadas (x, y, z) da última célula a atingir o valor $D \leq 10^{-3}$ e a última coluna, o valor do divergente atingido.

Vento	Iterações	Célula	Divergente
#1	399	(38,16,3)	1,00E-03
#2	116	(39,16,2)	9,79E-04
#3	976	(38,16,2)	9,99E-04
#4	1668	(38,16,4)	9,99E-04
#5	113	(37,10,2)	9,99E-04
#6	235	(41,29,2)	9,95E-04
#7	138	(38,16,3)	9,98E-04
#8	431	(38,16,3)	9,98E-04
#9	1103	(38,16,3)	1,00E-03
#10	232	(38,16,3)	9,98E-04
#11	106	(37,15,2)	9,85E-04
#12	138	(46,15,1)	9,92E-04
#13	1188	(38,16,2)	1,00E-03
#14	1802	(38,16,2)	1,00E-03
#15	173	(38,15,2)	9,88E-04
#16	154	(41,30,1)	9,95E-04
#17	166	(38,15,2)	9,95E-04
#18	214	(38,15,2)	9,96E-04
#19	1304	(38,16,2)	1,00E-03
#20	171	(38,15,2)	9,90E-04

Tabela 14 – Valores de cada vento observado até atingir $D \leq 10^{-3}$

Observa-se que na maioria das simulações o número de iterações necessárias para atingir os critérios de parada ($D \leq 10^{-3}$) é muito superior a 56 (usado no programa original), o que significa que provavelmente usando 56 iterações levou a simulação a parar prematuramente, com altos valores de velocidade da divergência. O exemplo, pode ser visto na Figura 51, que mostra a evolução da velocidade da divergência da última célula que atingiu $D \leq 10^{-3}$ na experiência com o vento #14 (1802 iterações). Note, que com 56 iterações, a divergência é relativamente muito alta.

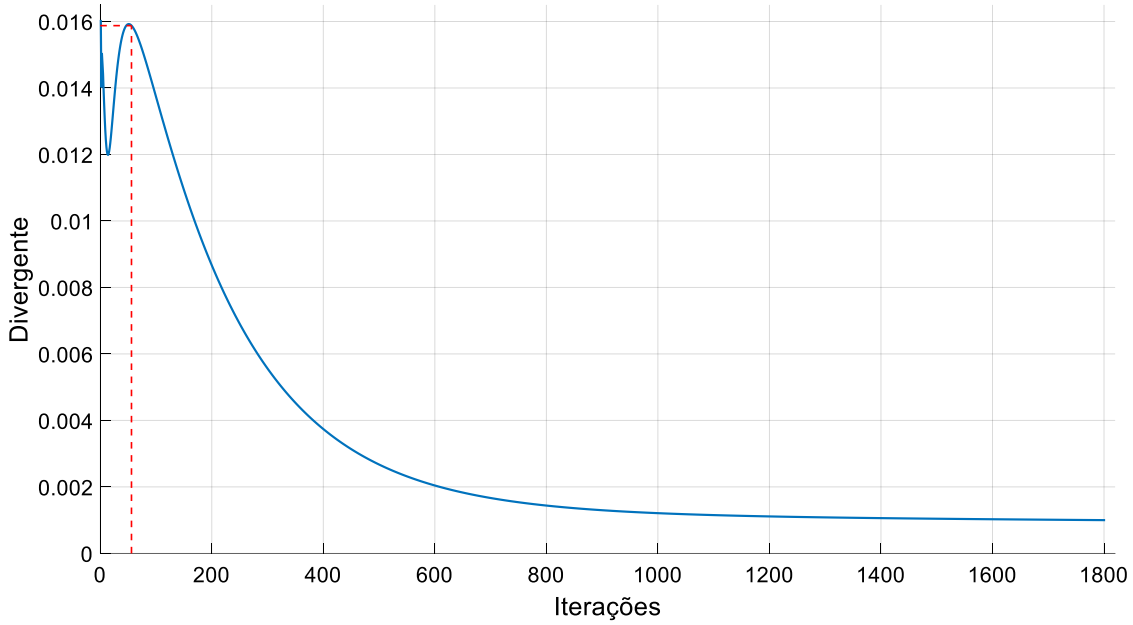

Figura 51 – Evolução da última célula até atingir D ≤ 10^{-3} com vento #14

Em tais experimentos, conforme visto na Tabela 14 observa-se que o número de iterações até atingir o critério de parada apresenta uma grande variação de uma simulação para outra e as últimas células a convergirem ficam em uma região próxima para todos os experimentos. Uma causa para tais variações são os campos de vento iniciais (interpolados), que podem ser mais ou menos divergentes, de acordo com as características do vento observado (velocidade, direção, temperatura, estabilidade) e as características do terreno a favor do vento.

Outro fator observado que aumenta o número de iterações é que devido às características do vento e do terreno, algumas células tem o seu divergente sendo minimizado a uma taxa muito menor que o restante das demais células como no caso apresentado na Figura 51. Na Figura 52 é apresentado um experimento realizado com o vento #14 (1802 iterações) onde observa-se a evolução da primeira célula a atingir $D \leq 10^{-3}$.

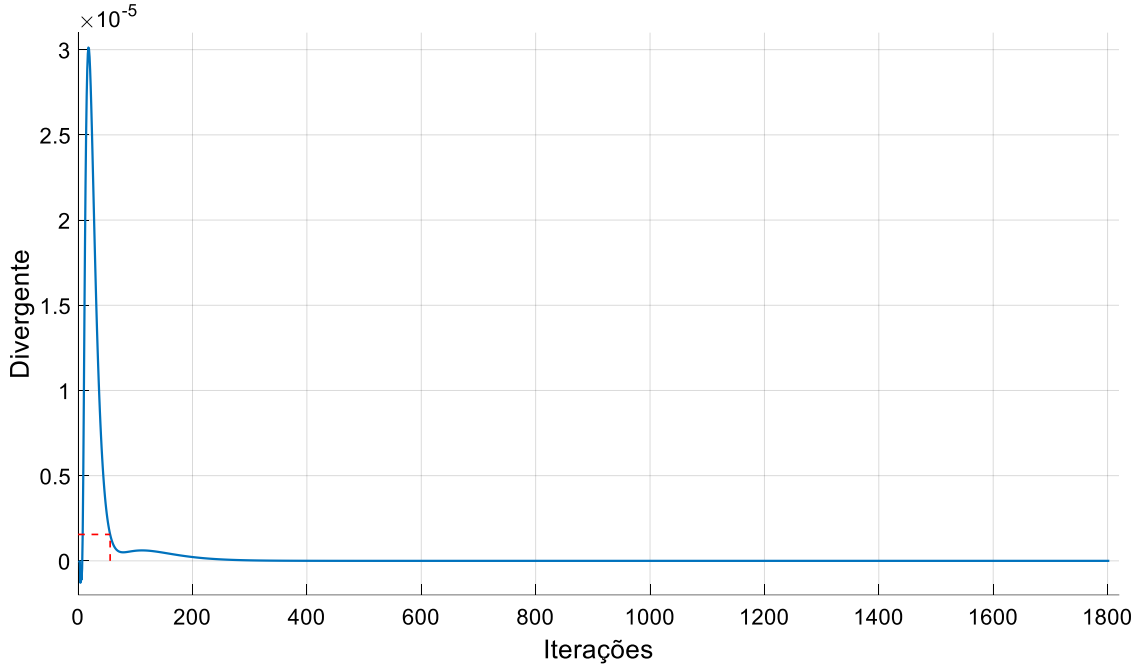

Figura 52 - Evolução da primeira célula até atingir D ≤ 10^{-3} com vento #14

Nesta simulação, para alcançar D ≤ 10^{-3}, o tempo de execução foi de 0,78 segundos, contrastando com os 0,04 segundos necessários para 56 iterações. Na verdade, a sobrecarga de tempo não tem grande impacto sobre o tempo computacional geral. Contudo, como se verá adiante, essa sobrecarga temporal torna-se muito considerável à medida que o domínio computacional é refinado.

3.4. PARALELISMO DAS FUNCÕES DE MENOR COMPLEXIDADE

As funções Extrapolacao_Vertical, Adicao_do_Terreno e Velocidade_Zero são funções que conforme visto na Tabela 6 representam custo computacional insignificante e não serão paralelizadas neste trabalho.

Já as funções Interpolacao_da_Estabilidade, Inicializacao_do_Campo_de_Vento, Interpolacao_da_Velocidade, Calculo_da_Transparencia e Remocao_da_Divergencia, possuem um certo custo computacional e a paralelização destas funções contribuem para

a redução do tempo total de processamento, entretanto, a paralelização destas funções é bastante simples onde o cálculo de cada célula independe de outras células, portanto, o paralelismo é direto e na grande maioria das vezes utilizando-se as técnicas já descritas.

3.4.1. Algoritmo paralelo da função Interpolação da Estabilidade

O desenvolvimento do algoritmo paralelo da função Interpolação da Estabilidade em sua versão paralela possui baixa complexidade de paralelismo pois, o cálculo de cada célula independe do cálculo de outra célula no mesmo ciclo. A implementação do kernel CUDA da função Interpolacao_da_Estabilidade_Kernel pode ser visto na Figura 53.

```
__global__ void Interpolacao_da_Estabilidade_Kernel (parametros){
 //Declarações e inicializações
 ...
 // Calcula os índices da matriz usando a identificação do thread com Grid-Stride Loop
 for (KK = blockIdx.z * blockDim.z + threadIdx.z; KK < NZ; KK += blockDim.z * gridDim.z){
  K = NZ - KK;
  for (I = blockIdx.y * blockDim.y + threadIdx.y; I < NX; I += blockDim.y * gridDim.y){
   for (J = blockIdx.x * blockDim.x + threadIdx.x; J < NY; J += blockDim.x * gridDim.x){
    // Calcula os índices em um grande vetor (matrizes 3D são transformadas em 1D)
    index_Estab = (J + I * NY) * NZ + K;
    index_Iht = J + I * NY;
    // Atribuição do valor zero para a estabilidade nas células obstáculo
    Estab[index_Estab]= 0.0;
    ...
    // Numero de torres de estabilidade válidas
    for (L = 1; L <= NumEsd; L++){
     // Calcula os índices em um grande vetor (matrizes 3D são transformadas em 1D)
     index_Ess = K + L * NZ;
     ...
     // Soma do quadrado dos lados
     RFOR = (pow (((Ies[L] - I) * DX), 2) + pow (((Jes[L] - J) * DY), 2));
     ...
     // Somatório da estabilidade da torre pelo quadrado da distância
     Estab[index_Estab]= Estab[index_Estab]+ Ess[index_Ess]/ RFOR;
     // Somatório do inverso do quadrado da distância
     XNORM = XNORM + 1.0 / RFOR;
    }
    ...
    //Ponderação das células interpoladas
    if (XNORM > 0)
     Estab[index_Estab]= Estab[index_Estab]/ XNORM;
    ...
   } // I
  } // J
 } // K
}
```

Figura 53 – Algoritmo paralelo da função Interpolacao_da_Estabilidade_Kernel

A Figura 54 mostra a versão paralela GPU / CUDA da função Interpolacao_da_Estabilidade. A escolha da quantidade de threads por bloco de 8 nas dimensões X, Y e Z será explicada adiante.

```
void Interpolacao_da_Estabilidade (parametros){
   //Declarações e inicializações
   ...
   // Dimensão do Grid e Bloco
   Dim_Grid_X = NX / 8;   // 8 = Threads por bloco em X
   if ((NX % 8) != 0) // 8 = Threads por bloco em X
      Dim_Grid_X++;
   // Repete para Y and Z
   ...
   DimGrid(Dim_Grid_X, Dim_Grid_Y, Dim_Grid_Z);
   DimBlock(8, 8, 8); // DimBlock(Threads X, Threads Y, Threads Z)
   // Alocação de memória na GPU
   ...
   // Copia dados para a GPU
   ...
   //Inicia função no Kernel
   Interpolacao_da_Estabilidade_Kernel << <DimGrid, DimBlock >> > (parametros);
   // Sincroniza threads
   cudaDeviceSynchronize();
   // Copia resultados da GPU
   ...
   // Liberação de memória e finalização CUDA
   ...
}
```

Figura 54 – Chamada do kernel da função Interpolação da Estabilidade

3.4.2. Algoritmo paralelo da função Inicialização do Campo de Vento

O desenvolvimento do algoritmo paralelo da função Inicialização do Campo de Vento em sua versão paralela possui baixa complexidade, nesta implementação foi utilizada a técnica monolitic kernel pois, devido a sua simplicidade não existe a necessidade da técnica grid-stride loop. A implementação do kernel CUDA da função Inicializacao_do_Campo_de_Vento_Kernel pode ser visto na Figura 55.

```
__global__ void Inicializacao_do_Campo_de_Vento_Kernel (parametros){
   //Declarações e inicializações
   ...
   // Calcula os índices da matriz usando Monolitic Kernel
   K = blockIdx.z * blockDim.z + threadIdx.z;
   I = blockIdx.y * blockDim.y + threadIdx.y;
   J = blockIdx.x * blockDim.x + threadIdx.x;
   // Calcula os índices em um grande vetor (matrizes 3D são transformadas em 1D)
   index_VentoU = (J + I * NY) * NZ + K;
   ...
   if ((K != NZ + 1) && (I != NX + 1))
      VENTOU[index_VentoU] = 0.0;
   if ((K != NZ + 1) && (J != NY + 1))
      VENTOV[index_VentoV] = 0.0;
   if ((I != NX + 1) && (J != NY + 1))
      VENTOW[index_VentoW] = 0.0;
}
```

Figura 55 – Algoritmo paralelo da função Inicializacao_do_Campo_de_Vento_Kernel

A Figura 56 mostra a versão paralela GPU / CUDA da função Inicializacao_do_Campo_de_Vento.

```
void Inicializacao_do_Campo_de_Vento (parametros){
   //Declarações e inicializações
   ...
   // Dimensão do Grid e Bloco
   ...
   // Alocação de memória na GPU
   ...
   // Copia dados para a GPU
   ...
   //Inicia função no Kernel
   Inicializacao_do_Campo_de_Vento_Kernel << <DimGrid, DimBlock >> > (parametros);
   // Sincroniza threads
   ...
   // Copia resultados da GPU
   ...
   // Liberação de memória e finalização CUDA
   ...
}
```

Figura 56 – Chamada do kernel da função Inicialização do Campo de Vento

3.4.3. Algoritmo paralelo da função Interpolação da Velocidade

O desenvolvimento do algoritmo paralelo da função Interpolação da Velocidade em sua versão paralela possui baixa complexidade. A implementação do kernel CUDA da função Inicializacao_do_Campo_de_Vento_Kernel pode ser visto na Figura 57.

```
__global__ void Interpolacao_da_Velocidade_Kernel (parâmetros){
 //Declarações e inicializações
 ...
 // Calcula os índices da matriz usando a identificação do thread com Grid-Stride Loop
 for (I = blockIdx.y * blockDim.y + threadIdx.y; I < NX + 1; I += blockDim.y * gridDim.y){
  for (J = blockIdx.x * blockDim.x + threadIdx.x; J < NY + 1; J += blockDim.x * gridDim.x){
   for (K = blockIdx.z * blockDim.z + threadIdx.z; K < NZ; K += blockDim.z * gridDim.z){
    // Calcula os índices em um grande vetor (matrizes 3D são transformadas em 1D)
    index_VentoU = (J + I * NY ) * NZ + K);
    ...
    // Numero de torres de estabilidade válidas
    for (L = 1; L <= NumEwd; L++){
     // Calcula os índices em um grande vetor (matrizes 3D são transformadas em 1D)
     indexW = K  + L * NZ;
     // Cálculo do quadrado da distância na coordenada X e Y
     RSQDX = pow ((Iew[L] - I + 0.5), 2) * DX2 + pow ((Jew[L] - J), 2) * DY2;
     RSQDY = pow ((Iew[L] - I), 2) * DX2 + pow ((Jew[L] - J + 0.5), 2) * DY2;
     //Somatorio da componente da velocidade na torre L, fator de confiança e
       inverso do quadrado da distância
     if (J != NY + 1)
      VENTOU[index_VentoU]= VENTOU[index_VentoU]+ WU[indexW]* Eww[indexW]/ RSQDX;
     if (I != NX + 1)
      VENTOV[index_VentoV]= VENTOV[index_VentoV]+ WV[indexW]* Eww[indexW]/ RSQDY;
     // Somatório do fator de confiança da torre L
     XNORMA = XNORMA + Eww[L][K] / RSQDX;
     YNORMA = YNORMA + Eww[L][K] / RSQDY;
    }
    // Média das NumEwd torres para cada célula
    if (J != NY + 1)
     VENTOU[index_VentoU]= VENTOU[index_VentoU]/ XNORMA;
    if (I != NX + 1)
     VENTOV[index_VentoV]= VENTOV[index_VentoV]/ YNORMA;
   } // K
  } // J
 } // I
}
```

Figura 57 – Algoritmo paralelo da função Interpolacao_da_Velocidade_Kernel

A Figura 58 mostra a versão paralela GPU / CUDA da função Interpolacao_da_Velocidade.

```
void Interpolacao_da_Velocidade (parametros){
   //Declarações e inicializações
   ...
   // Dimensão do Grid e Bloco
   ...
   // Alocação de memória na GPU
   ...
   // Copia dados para a GPU
   ...
   //Inicia função no Kernel
   Interpolacao_da_Velocidade_Kernel << <DimGrid, DimBlock >> > (parametros);
   // Sincroniza threads
   ...
   // Copia resultados da GPU
   ...
   // Liberação de memória e finalização CUDA
   ...
}
```

Figura 58 – Chamada do kernel da função Interpolação da Velocidade

3.4.4. Algoritmo paralelo da função Cálculo da Transparência

O desenvolvimento do algoritmo paralelo da função Cálculo da Transparência em sua versão paralela é aqui apresentado. A implementação do kernel CUDA da função Calculo_da_Transparência_Kernel pode ser visto na Figura 59.

```
__global__ void Calculo_da_Transparencia_Kernel (paremetros){
 //Declarações e inicializações
 ...
 // Calcula os índices da matriz usando a identificação do thread com Grid-Stride Loop
 for (int I = blockIdx.y * blockDim.y + threadIdx.y; I < NX; I += blockDim.y * gridDim.y){
  for (int J = blockIdx.x * blockDim.x + threadIdx.x; J < NY; J += blockDim.x * gridDim.x){
   for (int K = blockIdx.z * blockDim.z + threadIdx.z; K < NZ; K += blockDim.z * gridDim.z){
    // Calcula os índices em um grande vetor (matrizes 3D são transformadas em 1D)
    index_Estab = (J + (I * NY)) * NZ + K;
    index_TX1   = (J + (1 * NY)) * NZ + K;
    ...
    // cálculo do coeficiente de transparência para a direção X
    ...
    // Verifica se é célula obstáculo
    if (Iht[index_Iht]< K){
     // Calcula o valor do coeficiente de transparência
     TX[index_TX1]= Calc([index_Estab], TH);
    }
    Else {
     // Valor zero para o coeficiente de transparência das células obstáculo
     TX[index_TX1]= 0.0;
    }
    ...
    // cálculo do coeficiente de transparência para a direção Y
    ...
    // cálculo do coeficiente de transparência para a direção Z
    ...
   } // K
  } // J
 } // I
`
```

Figura 59 – Algoritmo paralelo da função Calculo_da_Transparencia_Kernel

A Figura 60 mostra a versão paralela GPU / CUDA da função Calculo_da_Transparência.

```
void Calculo_da_Transparencia (parametros){
   //Declarações e inicializações
   ...
   // Dimensão do Grid e Bloco
   ...
   // Alocação de memória na GPU
   ...
   // Copia dados para a GPU
   ...
   //Inicia função no Kernel
   Calculo_da_Transparencia_Kernel << <DimGrid, DimBlock >> > (parametros);
   // Sincroniza threads
   ...
   // Copia resultados da GPU
   ...
   // Liberação de memória e finalização CUDA
   ...
}
```

Figura 60 – Chamada do kernel da função Cálculo da Transparência

3.4.5. Algoritmo paralelo da função Remoção da Divergência

O desenvolvimento do algoritmo paralelo da função Remoção da Divergência em sua versão paralela apresenta uma baixa complexidade em seu paralelismo. A implementação do kernel CUDA da função Remoção_da_Divergência_Kernel pode ser visto na Figura 61.

```
__global__ void Remocao_da_Divergencia_Kernel (parametros){
 //Declarações e inicializações
 ...
 // Calcula os índices da matriz usando a identificação do thread com Grid-Stride Loop
 for (int I = blockIdx.y * blockDim.y + threadIdx.y; I < NX; I += blockDim.y * gridDim.y){
  for (int J = blockIdx.x * blockDim.x + threadIdx.x; J < NY; J += blockDim.x * gridDim.x){
   for (int K = blockIdx.z * blockDim.z + threadIdx.z; K < NZ; K += blockDim.z * gridDim.z){
     // Calcula os índices em um grande vetor (matrizes 3D são transformadas em 1D)
     index_I = (J + I * NY) * NZ + K;
     index_IP1 = (J + IP1 * NY) * NZ + K;
     ...
     VENTOW[index_KP1] = (VENTOU[index_I] - VENTOU[index_IP1]) * DZ / DX +
                         (VENTOV[index_J] - VENTOV[index_JP1]) * DZ / DY +
                          VENTOW[index_K];
   } // K
  } // J
 } // I
}
```

Figura 61 – Algoritmo paralelo da função Remocao_da_Divergencia_Kernel

A Figura 62 mostra a versão paralela GPU / CUDA da função Remocao_da_Divergencia.

```
void Remocao_da_Divergencia (parametros){
   //Declarações e inicializações
   ...
   // Dimensão do Grid e Bloco
   ...
   // Alocação de memória na GPU
   ...
   // Copia dados para a GPU
   ...
   //Inicia função no Kernel
   Remocao_da_Divergencia_Kernel << <DimGrid, DimBlock >> > (parametros);
   // Sincroniza threads
   ...
   // Copia resultados da GPU
   ...
   // Liberação de memória e finalização CUDA
   ...
}
```

Figura 62 – Chamada do kernel da função Remoçao da Divergência

3.4.6. Resultados

Alguns resultados foram obtidos das funções de menor complexidade, entretanto, seus valores não apresentam uma melhora significativa com o paralelismo, pois os tempos calculados para as versões de GPU, incluem a execução de kernels na GPU, alocação de memória na GPU e transferência de dados para a GPU que para estas funções em domínios computacionais pequenos, acaba sendo maior que o tempo executado somente na CPU.

Algumas dessas funções foram descritas em investigações preliminares, relatadas em (PINHEIRO, DESTERRO, *et al.*, 2016) onde, um domínio computacional maior é utilizado e valores de *speedup* entre aproximadamente 10 e 40 foram obtidos. A Tabela 15 mostra os valores obtidos para as funções de menor complexidade de paralelismo para 2000 iterações utilizando o vento #2.

Nome da função	CPU (s)	GPU* (s)
Interpolacao_da_Estabilidade	0,002000000	0,096833333
Inicializacao_do_Campo_de_Vento	0,000833333	0,000166667
Interpolacao_da_Velocidade	0,013666667	0,000833333
Calculo_da_Transparencia	0,000833333	0,001000000
Remocao_da_Divergencia	0,001820988	0,002185185

*Considerando kernel GPU + alocação de memória na GPU + transferência de dados para a GPU

Tabela 15 – Tempo das funções de menor complexidade

3.5. PARALELISMO DAS FUNÇÕES DE MAIOR COMPLEXIDADE

A abordagem paralela para a Minimização da Divergência (função Minimizacao_da_Divergencia) não é simples. O algoritmo é iterativo e apresenta forte natureza sequencial devido à propagação dos sinais de correção de velocidade (para minimizar a divergência) para todo o domínio computacional, o que cria dependência

entre células contínuas. Além disso, é de longe a função que consome mais tempo de execução do programa (92,23% vide Tabela 6). Por estas razões, o principal foco aqui é descrever as questões mais importantes e a avaliação de desempenho da versão paralela da função Minimizacao_da_Divergencia. Para conseguir isso, iniciou-se observando o algoritmo do código sequencial da função Minimizacao_da_Divergencia, mostrado na Figura 49.

3.6. O ALGORITMO PARALELO DO CAMPO DE VENTO

Embora não complicado, o algoritmo de Minimização da Divergência pode ser muito demorado se for aplicada uma discretização de malha fina do domínio computacional. No entanto, o paralelismo é não-trivial, uma vez que existem dependências entre células contínuas devido à propagação do sinal. Fica explícito ao observar as equações (18) a (23), que a quantidade adicionada a uma dada célula é subtraída do seu vizinho (célula subsequente). Antes de implementar a versão paralela da função Minimização_da_Divergencia, duas observações importantes devem ser consideradas:

Observação # 1: o conceito de propagação do sinal é realmente importante, uma vez que é responsável pela acomodação e propagação da divergência local. No entanto, a maneira iterativa que é implementada impõe um forte processo sequencial (dependência de células vizinhas);

Observação # 2: componentes de velocidade de uma determinada célula não podem ser atualizadas ao mesmo tempo por 2 processos independentes (ou threads). No entanto, em cada iteração de loop, 4 células são atualizadas: (I, J, K), (I + 1, J, K), (I, J +

1, K) e (I, J, K + 1). Portanto, as instruções dentro do loop (especialmente as equações (18) a (23)) não podem ser executadas em paralelo para células contínuas.

3.7. PARTIÇÃO DO DOMÍNIO: 3D-RED-BLACK

3.7.1. Definição do Problema

O conceito de propagação de sinal envolve a troca de uma "quantidade de velocidade" entre células contínuas em uma dada iteração. A quantia que um recebe é a quantidade que outro perde. Portanto, devido à Observação # 1, pelo menos duas células devem ser consideradas na mesma etapa de tempo para caracterizar a propagação do sinal. Por outro lado, a Observação #2 proíbe a execução paralela das instruções de loop, especialmente as equações (18) a (23), para células contínuas.

3.7.2. O Método 3D-Red-Black

Para superar tais restrições, foi aplicado o método Red-Black (FREEMAN e PHIILIPS, 1992), (YAVNEH, 1995) para a decomposição do domínio. O método permite o processamento paralelo de subconjuntos do domínio global. Neste trabalho, foi definida uma estrutura tridimensional vermelho-preta (3D-Red-Black) de tal forma que as células,

em que a soma dos índices (I + J + K) for ímpar, serão consideradas pretas (Black), caso contrário, elas serão consideradas vermelhas (Red), como ilustrado na Figura 63.

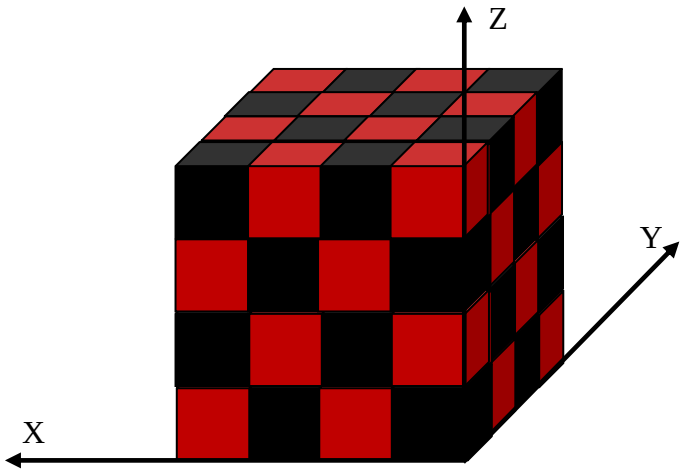

Figura 63 – Partição Tridimensional Red-Black

O algoritmo é executado em 2 passagens sequenciais (uma passagem Red e uma passagem Black). Na passagem Red, os índices I, J e K (em um dado passo) referem-se as células vermelhas. Assim, todas as células Red têm seus componentes de velocidade adicionados por uma certa quantidade (de acordo com as equações (18), (20) e (22)), enquanto seus vizinhos, as células Black, são subtraídas (equações (19), (21) e (23)). Na passagem Black, os índices I, J e K referem-se a células Black e o processo inverte: as células Black têm seus componentes adicionados por uma certa quantidade (equações (18), (20) e (22)), enquanto seus vizinhos, as células Red, são subtraídas (equações (19), (21) e (23)).

3.7.3. Versão sequencial do algoritmo 3D-Red-Black

Para analisar o comportamento da abordagem 3D-Red-Black proposta, antes de implementar a versão paralela (CUDA), foi desenvolvida uma versão sequencial 3D-Red-Black da função Minimizacao_da_ Divergencia. A Figura 64 mostra a função sequencial Minimizacao_da_ Divergencia_3D_Red_Black.

Diversos experimentos foram realizados com o algoritmo Minimizacao_da_Divergencia_3D-Red-Black, em que se demonstrou produzir resultados comparáveis aos obtidos pelo algoritmo original, com um padrão de convergência similar.

```
void Minimizacao_da_Divergencia_3D-Red-Black (parâmetros){
 //Declarações e inicializações
 ...
 while (!critério parada) {
  // Passo RED – Computa somente as células vermelhas (RED)
  for (K = 1; K <= NZ; K++){
   for (J = 1; J <= NY; J++){
    for (I = 1; I <= NX; I++){
     // Células RED
     if ((I+J+K) % 2 == 0) {
      // Coeficiente total de transissão
      TOT = 1.0 / ((T.X[I+1][J][K] + T.X[I][J][K]) / DX2 +
                   (T.Y[I][J+1][K] + T.Y[I][J][K]) / DY2 +
                   (T.Z[I][J][K+1] + T.Z[I][J][K]) / DZ2 );
      // Divergente
      D = (Vento.U[I+1][J][K] - Vento.U[I][J][K]) / DX +
          (Vento.V[I][J+1][K] - Vento.V[I][J][K]) / DY +
          (Vento.W[I][J][K+1] - Vento.W[I][J][K]) / DZ ;                          // Eq.(4)
      // Diferença da velocidade de perturbação
      delta_Phi = 1.25 * TOT * D;  // 1.25 = Fator de relaxação                  // Eq.(17)
      // Correção da velocidade
      Vento.U[I][J][K]   = Vento.U[I][J][K]   + delta_Phi * T.X[I][J][K]   / DX; // Eq.(18)
      Vento.U[I+1][J][K] = Vento.U[I+1][J][K] – delta_Phi * T.X[I+1][J][K] / DX; // Eq.(19)

      Vento.V[I][J][K]   = Vento.V[I][J][K]   + delta_Phi * T.Y[I][J][K]   / DY; // Eq.(20)
      Vento.V[I][J+1][K] = Vento.V[I][J+1][K] – delta_Phi * T.Y[I][J+1][K] / DY; // Eq.(21)

      Vento.W[I][J][K]   = Vento.W[I][J][K]   + delta_Phi * T.Z[I][J][K]   / DZ; // Eq.(22)
      Vento.W[I][J][K+1] = Vento.W[I][J][K+1] – delta_Phi * T.Z[I][J][K+1] / DZ; // Eq.(23)
     } // If
    } // I
   } // J
  } // K

  // Passo BLACK – Computa somente as células pretas (BLACK)
  for (K = 1; K <= NZ; K++){
   for (J = 1; J <= NY; J++){
    for (I = 1; I <= NX; I++){
     // Células BLACK
     if ((I+J+K) % 2 != 0) {
      // Coeficiente total de transissão
      TOT = 1.0 / ((T.X[I+1][J][K] + T.X[I][J][K]) / DX2 +
                   (T.Y[I][J+1][K] + T.Y[I][J][K]) / DY2 +
                   (T.Z[I][J][K+1] + T.Z[I][J][K]) / DZ2 );
      // Divergente
      D = (Vento.U[I+1][J][K] - Vento.U[I][J][K]) / DX +
          (Vento.V[I][J+1][K] - Vento.V[I][J][K]) / DY +
          (Vento.W[I][J][K+1] - Vento.W[I][J][K]) / DZ ;                          // Eq.(4)
      // Diferença da velocidade de perturbação
      delta_Phi = 1.25 * TOT * D;  // 1.25 = Fator de relaxação                  // Eq.(17)
      // Correção da velocidade
      Vento.U[I][J][K]   = Vento.U[I][J][K]   + delta_Phi * T.X[I][J][K]   / DX; // Eq.(18)
      Vento.U[I+1][J][K] = Vento.U[I+1][J][K] – delta_Phi * T.X[I+1][J][K] / DX; // Eq.(19)

      Vento.V[I][J][K]   = Vento.V[I][J][K]   + delta_Phi * T.Y[I][J][K]   / DY; // Eq.(20)
      Vento.V[I][J+1][K] = Vento.V[I][J+1][K] – delta_Phi * T.Y[I][J+1][K] / DY; // Eq.(21)

      Vento.W[I][J][K]   = Vento.W[I][J][K]   + delta_Phi * T.Z[I][J][K]   / DZ; // Eq.(22)
      Vento.W[I][J][K+1] = Vento.W[I][J][K+1] – delta_Phi * T.Z[I][J][K+1] / DZ; // Eq.(23)
     } // If
    } // I
   } // J
  } // K
  // Checa critério de convergência
  ...
 } // While
}
```

Figura 64 – Algoritmo sequencial da função Minimizacao_da_ Divergencia_3D-Red-Black

3.7.4. Validação da abordagem 3D-Red-Black

Antes da quantificação dos *speedups* e ganhos devido à paralelização, diversos experimentos foram realizados entre o algoritmo original e a abordagem 3D-Red-Black que demonstrou produzir resultados comparáveis aos obtidos pelo algoritmo original apresentando um padrão de convergência similar. O algoritmo proposto foi aplicado em simulações considerando todos os 20 ventos reais observados. A Tabela 16 mostra o número de iterações até o critério de parada ser obtido ($D \leq 10^{-3}$) para o algoritmo 3D-Red-Black e o algoritmo Original. Observa-se que o número de iterações é muito semelhante entre os algoritmos.

Vento	**3D-Red-Black**	**Original**
#1	427	399
#2	114	116
#3	925	976
#4	1429	1668
#5	112	113
#6	233	235
#7	147	138
#8	440	431
#9	1092	1103
#10	231	232
#11	111	106
#12	135	138
#13	1154	1188
#14	1687	1802
#15	178	173
#16	154	154
#17	165	166
#18	221	214
#19	1252	1304
#20	161	171
Média	518,40	541,35

Tabela 16 – Comparação entre a quantidade de iterações dos algoritmos

A Tabela 17 mostra a última célula a atingir o critério de parada ($D \leq 10^{-3}$) entre os algoritmos Original e 3D-Red-Black assim como, qual o valor do divergente alcançado nesta célula em ambos os algoritmos. Observa-se que tanto a célula quanto o valor do divergente alcançado apresentam grande coerência entre os algoritmos.

	Célula (X,Y,Z)		**Valor**	
Vento	**3D-Red-Black**	**Original**	**3D-Red-Black**	**Original**
#1	(38,16,3)	(38,16,3)	1,00E-03	1,00E-03
#2	(39,16,2)	(39,16,2)	9,97E-04	9,79E-04
#3	(38,16,2)	(38,16,2)	1,00E-03	9,99E-04
#4	(38,16,4)	(38,16,4)	9,99E-04	9,99E-04
#5	(37,10,2)	(37,10,2)	9,83E-04	9,99E-04
#6	(41,29,2)	(41,29,2)	9,90E-04	9,95E-04
#7	(38,16,3)	(38,16,3)	9,99E-04	9,98E-04
#8	(38,16,3)	(38,16,3)	9,98E-04	9,98E-04
#9	(38,16,3)	(38,16,3)	9,99E-04	1,00E-03
#10	(38,16,3)	(38,16,3)	9,98E-04	9,98E-04
#11	(37,15,2)	(37,15,2)	9,96E-04	9,85E-04
#12	(47,16,2)	(46,15,1)	9,99E-04	9,92E-04
#13	(38,16,2)	(38,16,2)	1,00E-03	1,00E-03
#14	(38,16,2)	(38,16,2)	1,00E-03	1,00E-03
#15	(38,15,2)	(38,15,2)	9,85E-04	9,88E-04
#16	(38,10,7)	(41,30,1)	9,90E-04	9,95E-04
#17	(38,15,2)	(38,15,2)	9,87E-04	9,95E-04
#18	(38,15,2)	(38,15,2)	9,93E-04	9,96E-04
#19	(38,16,2)	(38,16,2)	1,00E-03	1,00E-03
#20	(38,15,2)	(38,15,2)	9,95E-04	9,90E-04
Média			9,95E-04	9,95E-04

Tabela 17 – Comparação entre valores do divergente dos algoritmos

A Tabela 18 mostra as maiores diferenças (erro absoluto) entre as componentes (U, V e W) do campo de vento que foram calculadas pelos algoritmos Original e 3D-Red-Black, assim como a célula em que esta diferença ocorreu, para todos os ventos observados. Observa-se que alguma diferença aparece nas componentes de vento. Tais diferenças podem até ser consideradas, entretanto, elas são muito pequenas.

Vento	Célula (X,Y,Z)	Componente	Valor
#1	(39,16,2)	U	0,071160720
#2	(38,16,5)	U	0,008527220
#3	(39,16,2)	V	0,138697060
#4	(42,14,4)	W	0,479756000
#5	(34,16,5)	W	0,010210780
#6	(39,16,2)	V	0,090334290
#7	(14,19,3)	V	-0,066227700
#8	(39,15,3)	U	0,089708370
#9	(39,16,2)	V	0,058045400
#10	(39,15,3)	W	-0,101965000
#11	(38,15,2)	W	0,107991000
#12	(37,10,2)	U	-0,108301560
#13	(39,16,2)	U	0,065739040
#14	(39,16,2)	U	0,069904900
#15	(39,16,2)	V	0,126572645
#16	(38,16,8)	V	0,204585920
#17	(39,16,2)	W	-0,108210000
#18	(39,11,8)	U	-0,144946498
#19	(39,16,2)	W	0,082322100
#20	(37,16,2)	V	-0,078649890
Média			0,049762740

Tabela 18 – Componente com maior diferença entre os algoritmos

A Figura 65 mostra o vetor velocidade da célula com a maior diferença entre o algoritmo original (desenvolvido pelo PEN-COPPE/UFRJ) e do algoritmo 3D-Red-Black, na qual os vetores apresentam uma diferença de cerca de 2% em magnitude e menos de 0,007º em direção, demonstrando uma boa convergência entre os algoritmos original (Original) e 3D-Red-Black (3D-Red-Black).

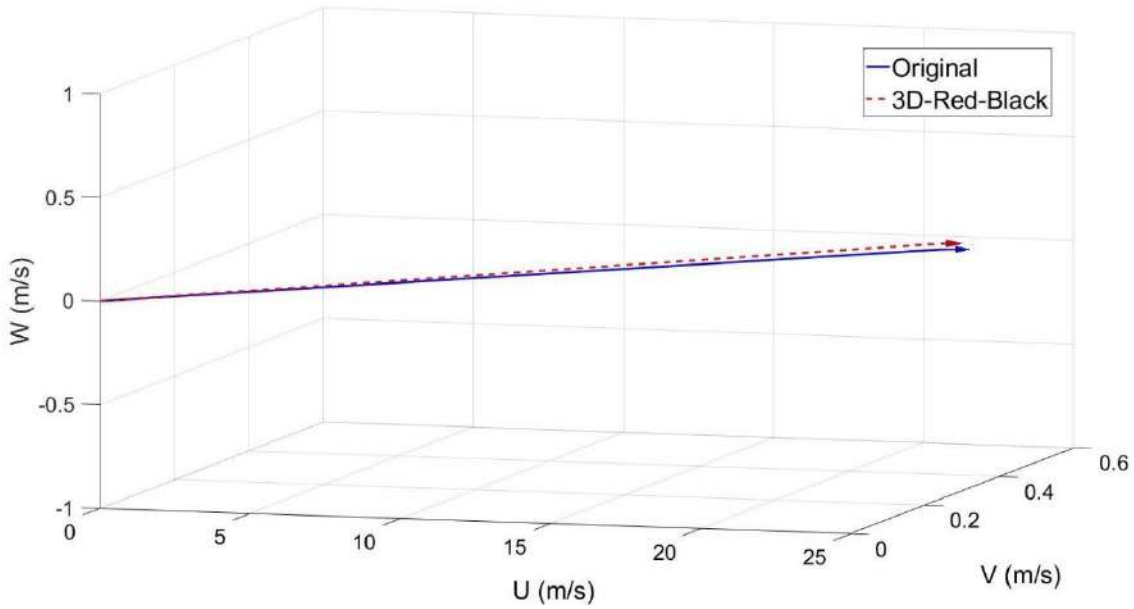

Figura 65 – Célula com a maior diferença entre os algoritmos original e 3D-Red-Black

A Figura 66 mostra uma comparação da convergência entre o divergente dos algoritmos original e 3D-Red-Black para uma célula típica. Note que é observado um comportamento semelhante do algoritmo 3D-Red-Black em relação ao algoritmo original.

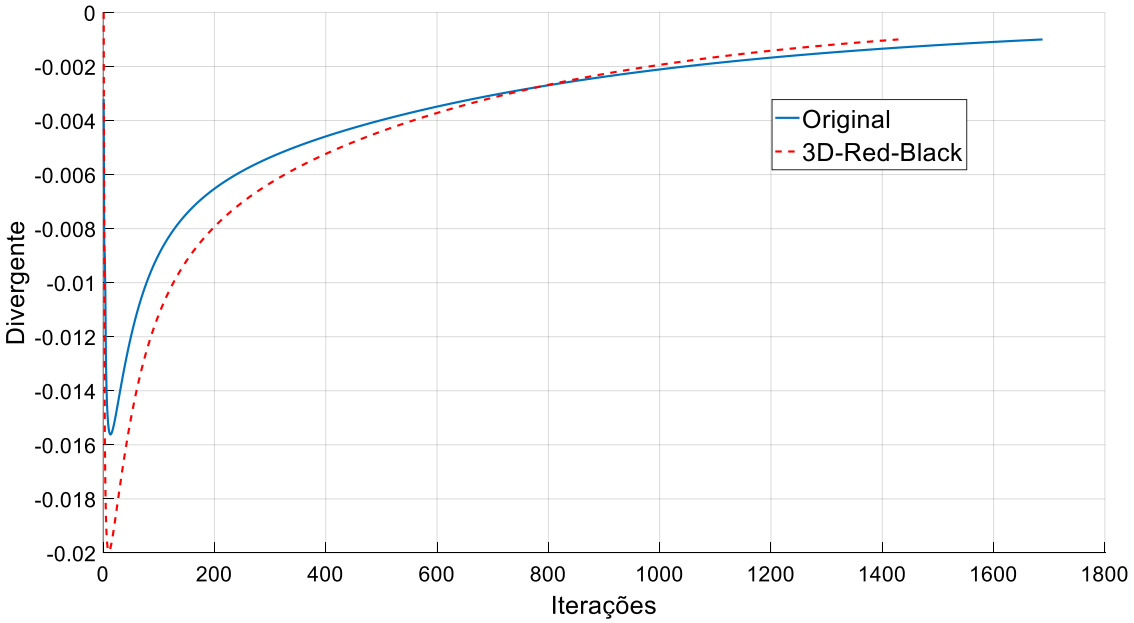

Figura 66 – Comparação da convergência do divergente entre os algoritmos original e 3D-Red-Black para uma célula típica

Resumindo, o algoritmo 3D-Red-Black é coerente com o algoritmo original, apresentando comportamentos e resultados semelhantes.

3.8. REFINAMENTO DO DOMÍNIO COMPUTACIONAL

Uma vez validada a abordagem sequencial 3D-Red-Black, foram aplicados refinamentos sucessivos nos domínios computacionais de forma a gerar topografias do terreno com maior resolução espacial para que, quando desenvolvida a versão CUDA paralela da minimização da divergência 3D-Red-Black, seu desempenho possa ser analisado.

Cada direção horizontal da célula original do domínio computacional foi sucessivamente subdividida, como mostrado na Figura 67. A dimensão vertical permaneceu inalterada.

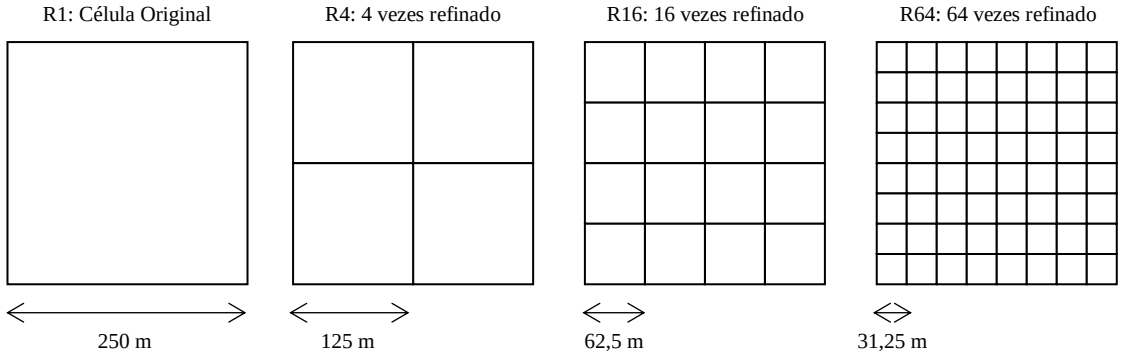

Figura 67 – Dimensões das células para diferentes níveis de refinamento

A Tabela 19 apresenta as dimensões da grade (número de partições de cada eixo) e o número total de células do domínio computacional para cada nível de refinamento.

	Dimensão			
Nível de refinamento	**X**	**Y**	**Z**	**Número de Células**
R1	67	43	8	23.048
R4	134	86	8	92.192
R16	268	172	8	368.768
R64	536	344	8	1.475.072

Tabela 19– Domínio computacional para diferentes níveis de refinamento

3.8.1. Convergência do modelo refinado

Pode ser observado que a convergência para domínios computacionais refinados é semelhante à original (R1), no entanto, são necessárias muito mais iterações para atingir o critério de parada. Para ilustrar isso, o comportamento da última célula a alcançar $D \leq 10^{-3}$, em simulações usando o Vento# 2, para todos os níveis de refinamento é apresentado nas figuras abaixo (Figura 68, Figura 69, Figura 70 e Figura 71).

A quantidade que o número de iterações aumenta com o refinamento do domínio computacional varia com diferentes ventos observados, no entanto, os padrões de convergência são muito semelhantes.

Note que o número de iterações até a convergência não apresenta diferença significativa entre implementações sequenciais e paralelas. Este foi um comportamento comum presente em todos os níveis de refinamento observados.

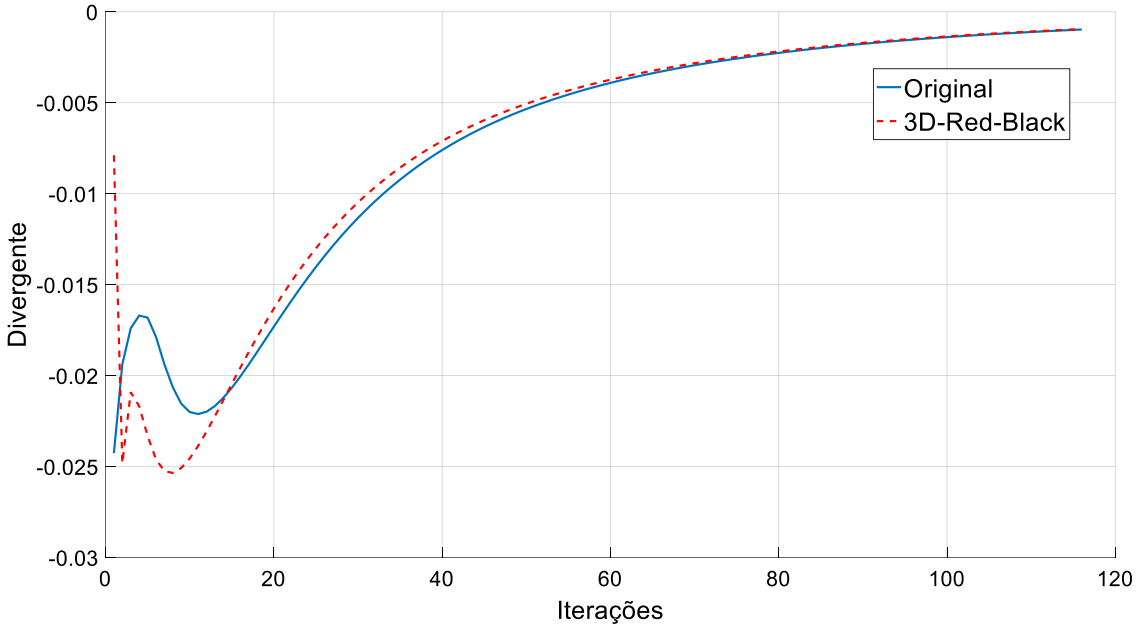

Figura 68 – Convergência do divergente para R1

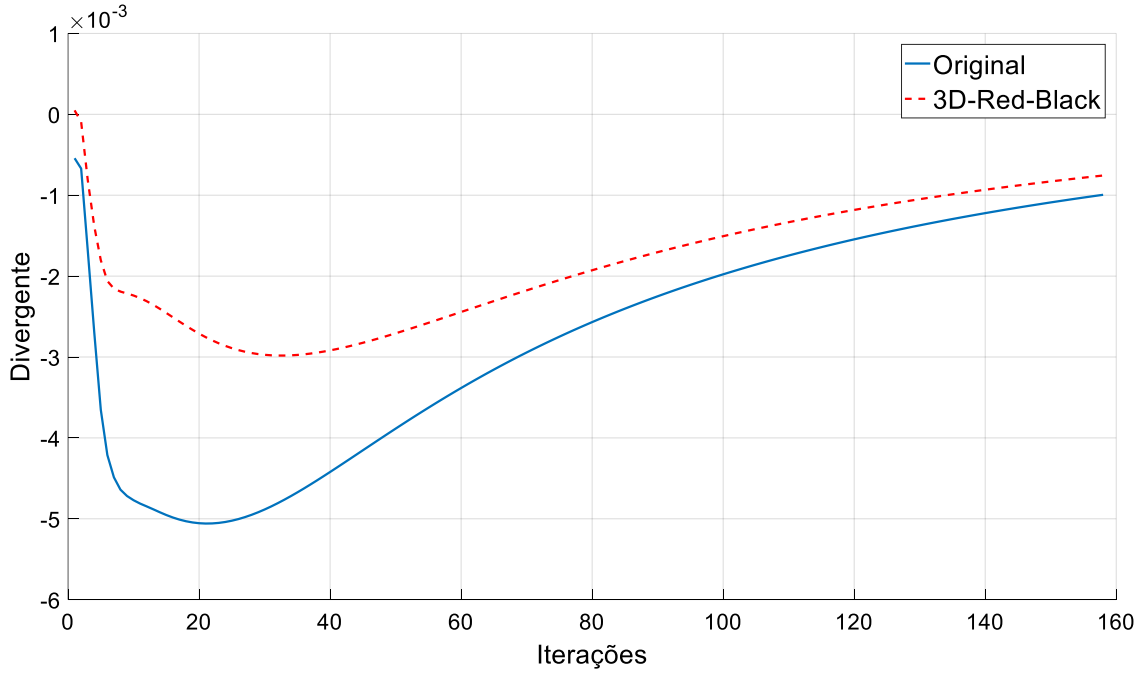

Figura 69 – Convergência do divergente para R4

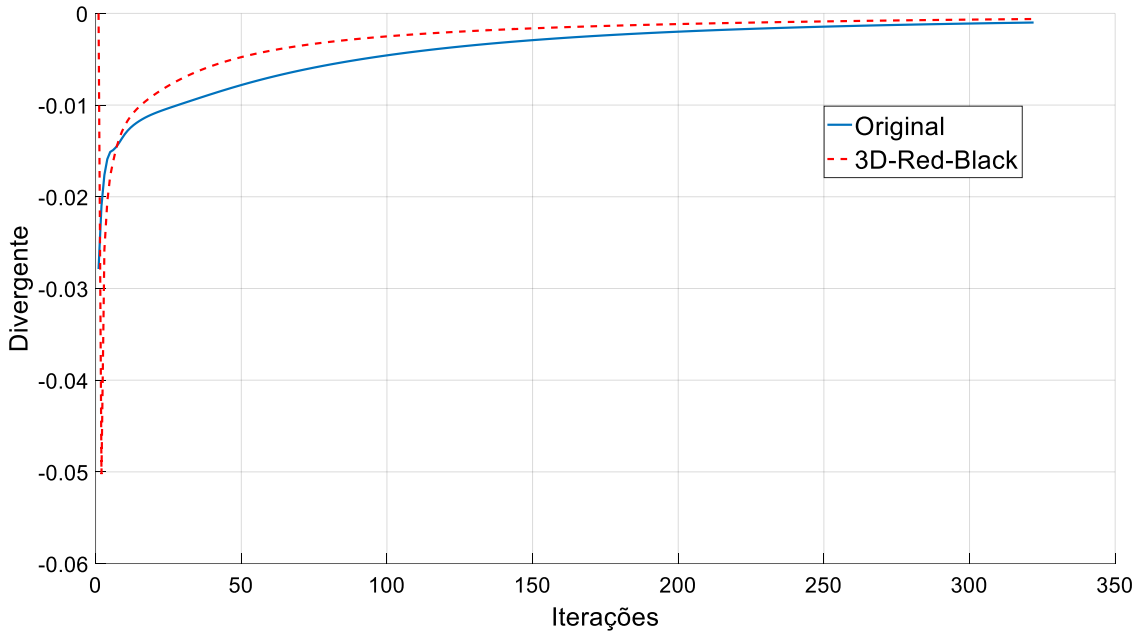

Figura 70 – Convergência do divergente para R16

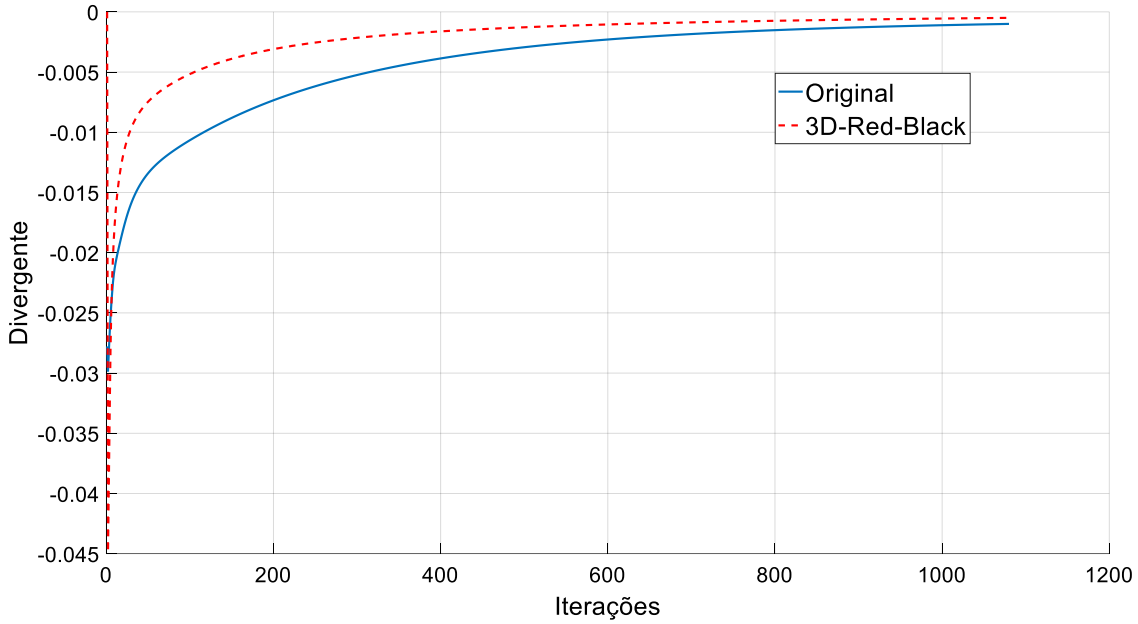

Figura 71 – Convergência do divergente para R64

Embora vários níveis de refinamento tenham sido investigados neste trabalho, o melhor refinamento a ser utilizado deve ser definido através de uma análise do sistema SCA como um todo, considerando a precisão das concentrações e doses previstas. No

entanto, como a versão refinada do módulo Dispersão da Pluma ainda está em desenvolvimento, tal investigação ocorrerá em trabalhos futuros.

3.9. PRIMEIRA IMPLEMENTAÇÃO PARALELA/CUDA DO 3D-RED-BLACK

3.9.1. Versão paralela/CUDA do algoritmo 3D-Red-Black

Depois de confirmado que a abordagem sequencial funcionou adequadamente, o desenvolvimento do algoritmo em sua versão paralela foi iniciado. A implementação do kernel CUDA da função Minimizacao_da_ Divergencia_3D_Red_Black_Kernel em sua primeira versão pode ser visto na Figura 72.

```
__global__ void Minimizacao_da_ Divergencia_3D_Red_Black_Kernel (parametros){
 //Declarações e inicializações
 ...
 // Calcula os índices da matriz usando a identificação do thread com Grid-Stride Loops
 for (int K = blockIdx.z * blockDim.z + threadIdx.z; K < NZ; K += blockDim.z * gridDim.z){
  for (int J = blockIdx.x * blockDim.x + threadIdx.x; J < NY; J += blockDim.x * gridDim.x){
   for (int I = blockIdx.y * blockDim.y + threadIdx.y; I < NX; I += blockDim.y * gridDim.y){
    if ((color == 'R') && ((I + J + K) % 2 != 0)) || // Passo RED
       ((color == 'B') && ((I + J + K) % 2 == 0)) { // Passo BLACK
     IP1 = I + 1;
     ...
     // Calcula os índices em um grande vetor (matrizes 3D são transformadas em 1D)
     index_I = (J + I * NY) * NZ + K;
     index_IP1 = (J + IP1 * NY) * NZ + K;
     ...
     // Coeficiente total de transissão
     TOT = 1.0 / (TX[index_IP1] + TX[index_I]) / DX2 +
                 (TY[index_JP1] + TY[index_J]) / DY2 +
                 (TZ[index_KP1] + TZ[index_K]) / DZ2;
     // Divergente
     D = (VENTOU[index_IP1] - VENTOU[index_I]) / DX +
         (VENTOV[index_JP1] - VENTOV[index_J]) / DY +
         (VENTOW[index_KP1] - VENTOW[index_K]) / DZ;                          // Eq.(4)
     // Diferença da velocidade de perturbação
     delta_Phi = 1.25 * TOT * D;  // 1.25 = Fator de relaxação                 // Eq.(17)
     // Correção da velocidade
     VENTOU[index_I]   = VENTOU[index_I]   + delta_Phi * TX[index_I]   / DX; // Eq.(18)
     VENTOU[index_IP1] = VENTOU[index_IP1] - delta_Phi * TX[index_IP1] / DX; // Eq.(19)

     VENTOV[index_J]   = VENTOV[index_J]   + delta_Phi * TY[index_J]   / DY; // Eq.(20)
     VENTOV[index_JP1] = VENTOV[index_JP1] - delta_Phi * TY[index_JP1] / DY; // Eq.(21)

     VENTOW[index_K]   = VENTOW[index_K]   + delta_Phi * TZ[index_K]   / DZ; // Eq.(22)
     VENTOW[index_KP1] = VENTOW[index_KP1] - delta_Phi * TZ[index_KP1] / DZ; // Eq.(23)
    } // If
   } // I
  } // J
 } // K
}
```

Figura 72 – Algoritmo paralelo da função Minimizacao_da_ Divergencia_3D_Red_Black_Kernel

Na função Minimizacao_da_ Divergencia_3D_Red_Black_Kernel (Figura 72), um grande grid de threads é gerenciado para processar a partição inteira (partição Black ou Red). Assim, os loops controlados pelas variáveis I, J e K (os loops que aparecem no algoritmo se referem a técnica grid-stride loop, já abordada) foram removidos e o kernel é chamado duas vezes: um para a passagem Red e outro para a passagem Black.

Outra consideração que proporcionou ganhos em termos de tempo computacional foi converter as matrizes 3D em um grande vetor (1D), concatenando cada dimensão. Neste caso, calcula-se uma correlação entre índices na matriz 3D e o índice correspondente no vetor. Observe, por exemplo, que U[index_JP1] corresponde a U[I][J+1][K], W[index_KP1] corresponde a W[I][J][K + 1] e assim por diante.

A Figura 73 mostra a versão paralela GPU / CUDA 3D-Red-Black da função Minimizacao_da_Divergencia_3D_Red_Black. Seguindo algumas recomendações encontradas na literatura de (WONG, PAPADOPOULOU, *et al.*, 2010), um ponto de partida seria a escolha de threads por blocos nas dimensões X, Y e Z de múltiplos de 2^n. Após várias experiências computacionais, utilizando-se combinações nas dimensões X, Y e Z dos múltiplos de 2^n, o melhor valor encontrado baseado no menor tempo de execução para a quantidade de threads por bloco foi de 8 nas dimensões X, Y e Z. Uma vez determinados, estes números foram utilizados para as duas primeiras experiências aqui descritas.

```
void Minimizacao_da_ Divergencia_3D_Red_Black (parametros){
   //Declarações e inicializações
   ...
   // Dimensão do Grid e bloco
   Dim_Grid_X = NX / 8;   // 8 = Threads por bloco em X
   if ((NX % 8) != 0) // 8 = Threads por bloco em X
      Dim_Grid_X++;
   // Repete para Y and Z
   ...
   DimGrid(Dim_Grid_X, Dim_Grid_Y, Dim_Grid_Z);
   DimBlock(8, 8, 8);  // DimBlock(Threads X, Threads Y, Threads Z)
   // Alocação de memória na GPU
   ...
   // Copia dados para a GPU
   ...
   //Inicia função no Kernel
  while (!critério parada) {
   // Passo RED – Computa somente as células vermelhas (RED)
   Minimizacao_da_ Divergencia_3D_Red_Black_Kernel <<<DimGrid, DimBlock>>>(parametros,'R');
   // Sincroniza threads
   cudaDeviceSynchronize();
   // Passo BLACK – Computa somente as células pretas (BLACK)
   Minimizacao_da_ Divergencia_3D_Red_Black_Kernel <<<DimGrid, DimBlock>>>(parametros,'B');
   // Sincroniza threads
   cudaDeviceSynchronize();
   // Checa critério de convergência
   ...
  } // while
  // Copia resultados da GPU
  ...
  // Liberação de memória e finalização CUDA
  ...
}
```

Figura 73 – Chamada do kernel da função Minimizacao da Divergencia 3D-Red-Black

3.9.2. Consistência da abordagem

Antes da quantificação dos *speedups* e ganhos devido à paralelização, diversos experimentos foram realizados entre o algoritmo sequencial e o algoritmo paralelo que, demonstrou produzir resultados comparáveis aos obtidos pelo algoritmo sequencial, apresentando valores praticamente idênticos. O algoritmo proposto foi aplicado em simulações considerando todos os 20 ventos reais observados. A Tabela 20 mostra as maiores diferenças entre as componentes (U, V e W) dos algoritmos sequencial e paralelo, assim como a célula em que esta diferença ocorreu, para todos os ventos observados.

Observa-se que uma diferença desprezível aparece nas componentes de vento o que era esperado, pois segundo o trabalho de (WHITEHEAD e FIT-FLOREA, 2011), as GPUs NVIDIA diferem da arquitetura x86, pois os modos de arredondamento nas GPUs são codificados em cada instrução de ponto flutuante ao contrário das CPUs onde a precisão é controlada dinamicamente pela palavra de controle de ponto flutuante (*floating point control word*). Em outras palavras, os algoritmos de paralelização reorganizam as operações, desta forma, gerando arredondamentos diferentes entre CPU e GPU e por consequência produzindo diferentes resultados numéricos.

Vento	**Célula (X,Y,Z)**	**Componente**	**Valor**
#1	(39,15,3)	V	2,01E-12
#2	(37,14,4)	U	2,11E-13
#3	(38,16,3)	V	3,65E-12
#4	(42,15,4)	U	1,89E-11
#5	(34,17,5)	U	3,31E-13
#6	(38,16,2)	V	-1,10E-11
#7	(15,19,4)	U	2,30E-12
#8	(39,15,3)	U	4,96E-13
#9	(39,15,5)	V	-2,61E-12
#10	(39,15,3)	U	4,24E-12
#11	(37,12,3)	U	5,12E-12
#12	(40,17,4)	V	1,08E-11
#13	(38,14,3)	V	9,97E-12
#14	(39,16,2)	V	3,69E-11
#15	(39,16,2)	V	3,50E-13
#16	(36,18,7)	W	-2,24E-11
#17	(37,13,5)	V	-1,29E-11
#18	(39,12,8)	W	2,08E-11
#19	(40,15,3)	V	-2,94E-12
#20	(38,14,3)	V	-2,63E-12
Média			6,16E-11

Tabela 20 – Componente com maior diferença entre os algoritmos

3.9.3. Resultados

Considerando que o algoritmo paralelo já está consistente, o objetivo desta seção é apenas quantificar e analisar os tempos de execução dos *speedups* obtidos com o uso do programa baseado em GPU. Para isso, foi escolhido um único vento observado (neste caso, Vento #2) e simulações com todos os níveis de refinamento (R1, R4, R16 e R64) e vários números de iterações (500, 1000, 1500 e 2000) foram investigados.

A Tabela 21 mostra os resultados comparativos entre os tempos de execução (em segundos) da rotina WEST do módulo de Campo de Vento (média de 6 execuções) para as implementações sequenciais (CPU) e paralelo (GPU_1) para os diferentes níveis de refinamento e número de iterações. Na implementação sequencial foi utilizada uma CPU Intel-I7 2700K @ 3.50GHz 3.90GHz e para o algoritmo paralelo, foi utilizada a mesma CPU com uma GPU GTX-680. Os tempos calculados para as versões de GPU incluem a execução de kernels na GPU, alocação de memória na GPU e transferência de dados para a GPU.

500			
	CPU	**GPU_1***	**$Speedup_1$**
R1	0,52	0,25	2,04
R4	3,00	0,46	6,55
R16	24,91	1,30	19,07
R64	112,24	4,66	24,09
1000			
	CPU	**GPU_1***	**$Speedup_1$**
R1	1,02	0,40	2,56
R4	5,94	0,81	7,34
R16	49,74	2,47	20,13
R64	223,68	9,09	24,62
1500			
	CPU	**GPU_1***	**$Speedup_1$**
R1	1,52	0,51	2,96
R4	10,02	1,16	8,63
R16	74,67	3,65	20,45
R64	334,67	13,52	24,76
2000			
	CPU	**GPU_1***	**$Speedup_1$**
R1	2,02	0,66	3,05
R4	13,52	1,51	8,95
R16	98,76	4,82	20,49
R64	449,29	18,04	24,91

*Considerando kernel GPU + alocação de memória na GPU + transferência de dados para a GPU

Tabela 21 - *Speedups* e tempos de execução da implementação sequencial e paralela

A Figura 74 até a Figura 77 nos mostram a influência do número de iterações no tempo de execução para todos os domínios computacionais.

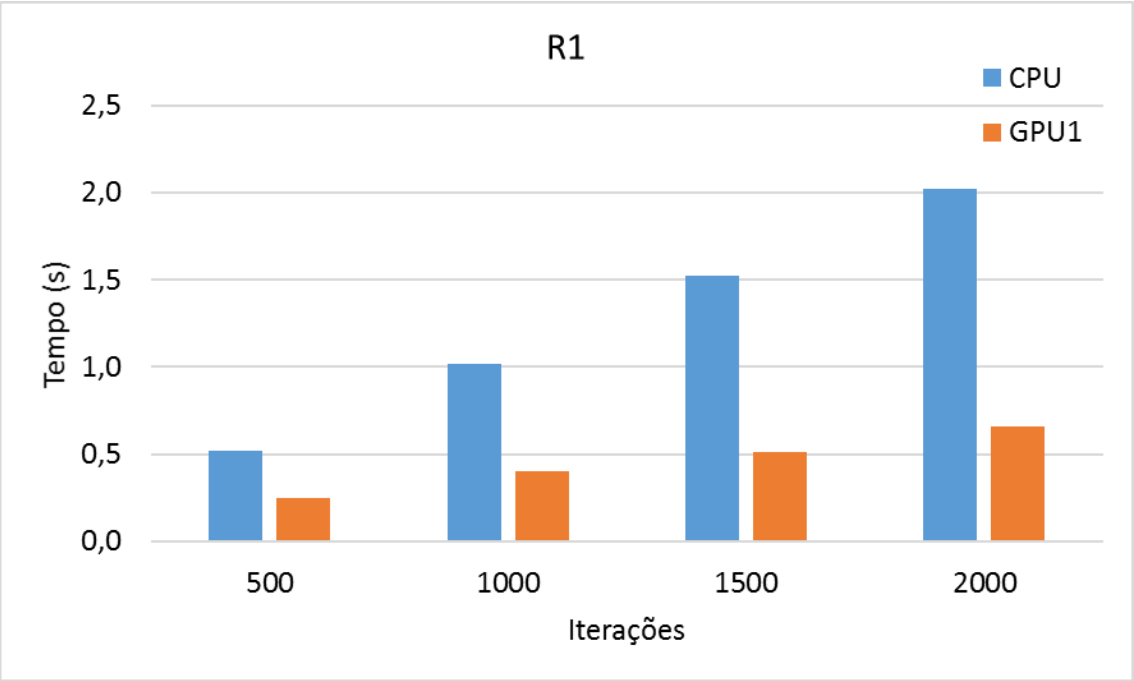

Figura 74 – Tempo de execução versus número de iterações (R1)

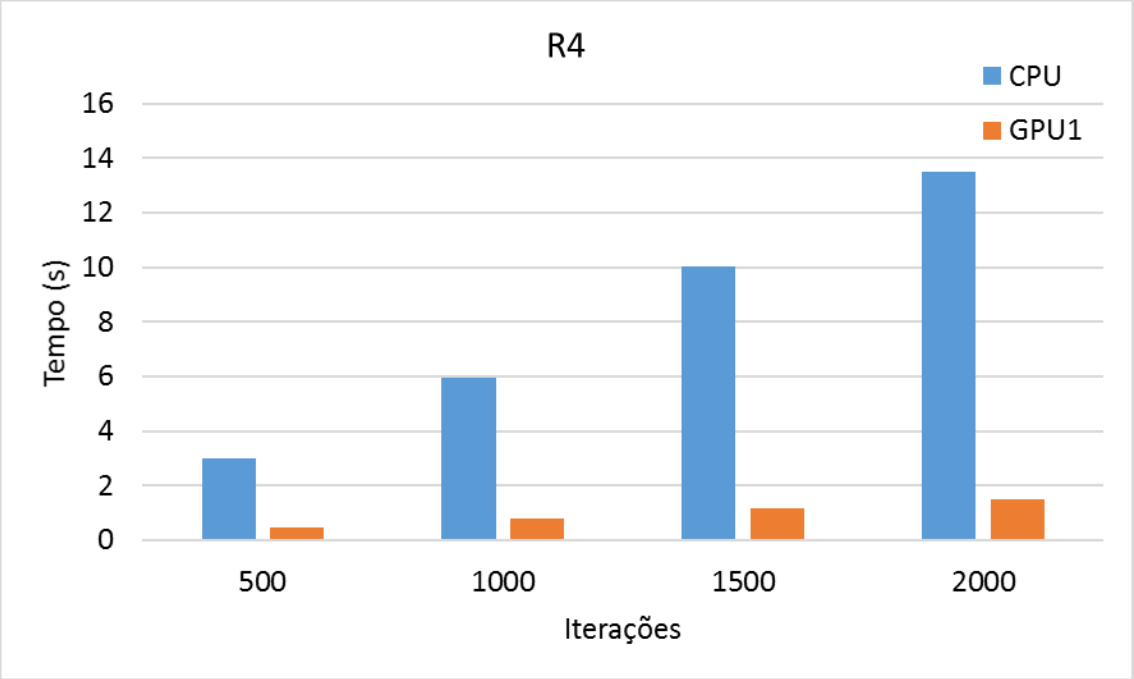

Figura 75 – Tempo de execução versus número de iterações (R4)

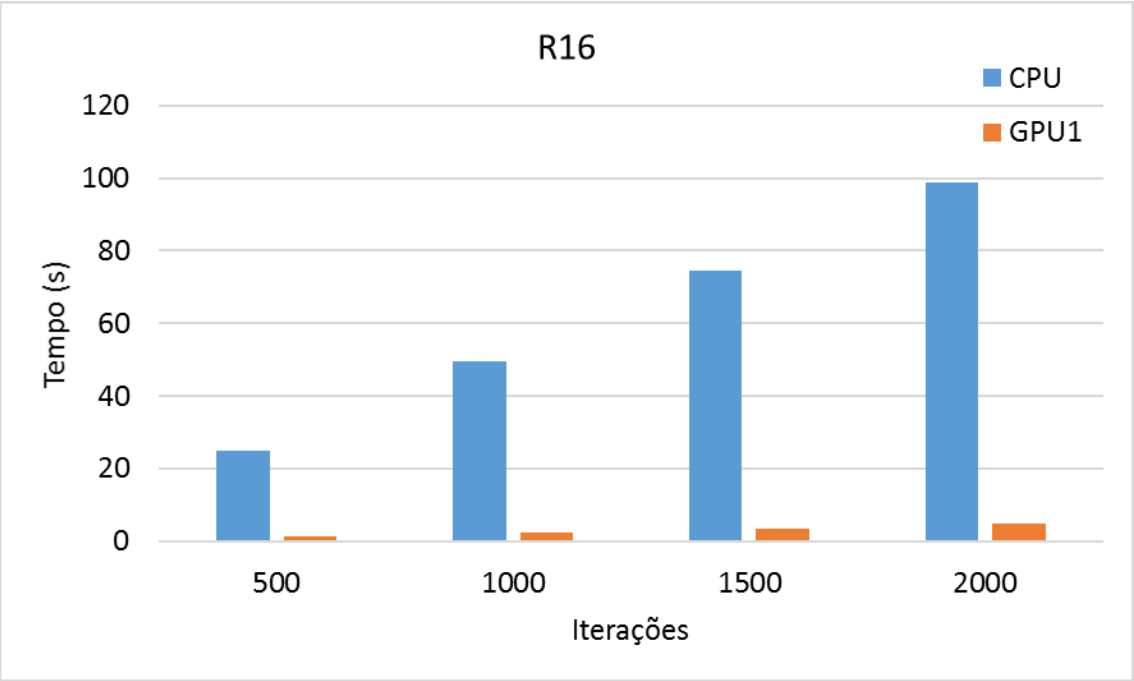

Figura 76 – Tempo de execução versus número de iterações (R16)

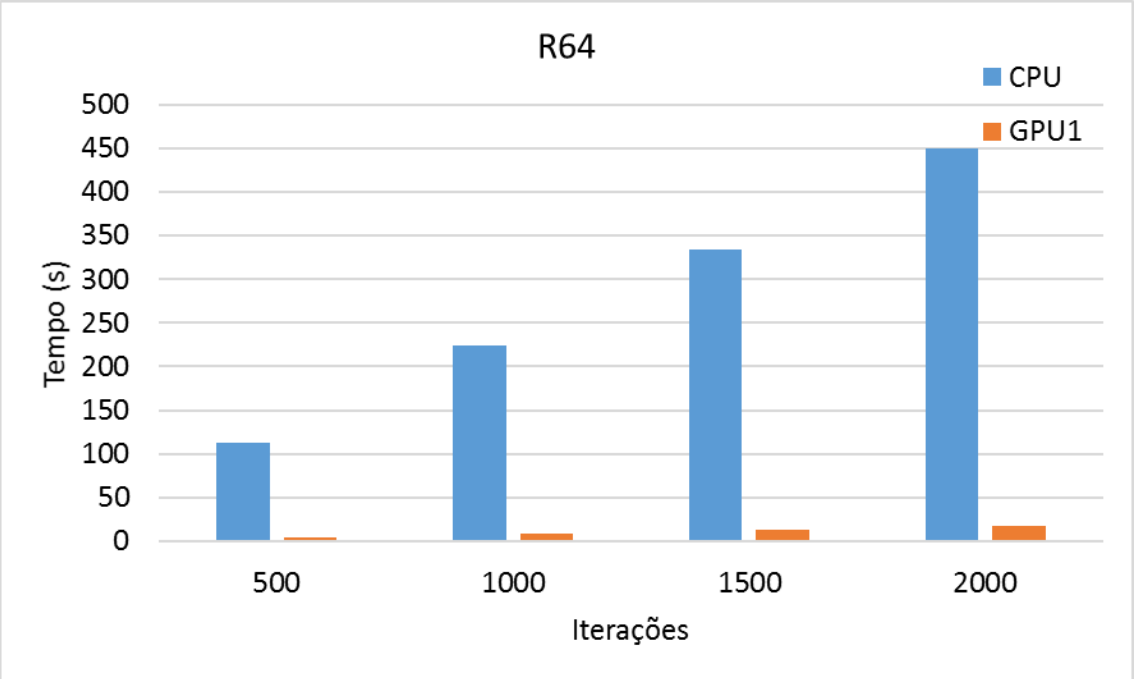

Figura 77 – Tempo de execução versus número de iterações (R64)

Como esperado, o tempo de execução é aproximadamente proporcional ao número de iterações. Isso ocorre porque a função Minimização_da_Divergência leva

mais de 92% do tempo de execução total do programa de Campo de Vento. Uma pequena diferença no tempo de execução, é devido as outras funções.

O *speedup*, consequentemente, é significativamente reduzido para simulações com menos iterações e domínios computacionais menores, onde se enfatiza a contribuição de outras funções no tempo total, conforme pode ser visto da Figura 78 até a Figura 81.

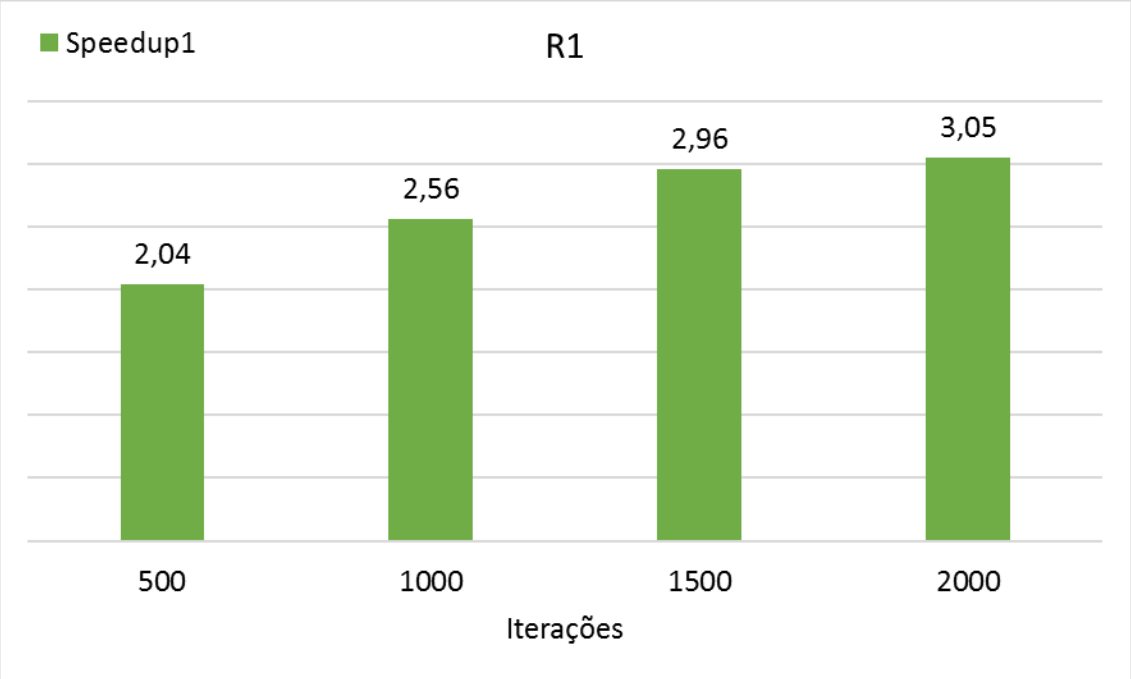

Figura 78 – *Speedup* versus número de iterações (R1)

Figura 79 – *Speedup* versus número de iterações (R4)

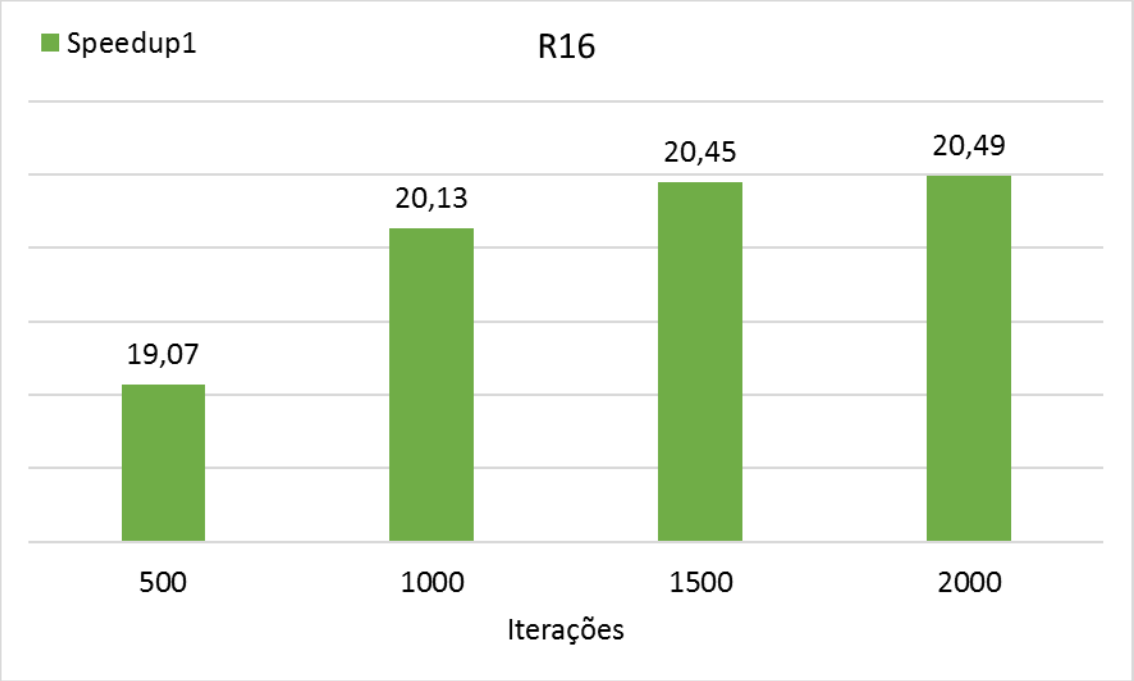

Figura 80 – *Speedup* versus número de iterações (R16)

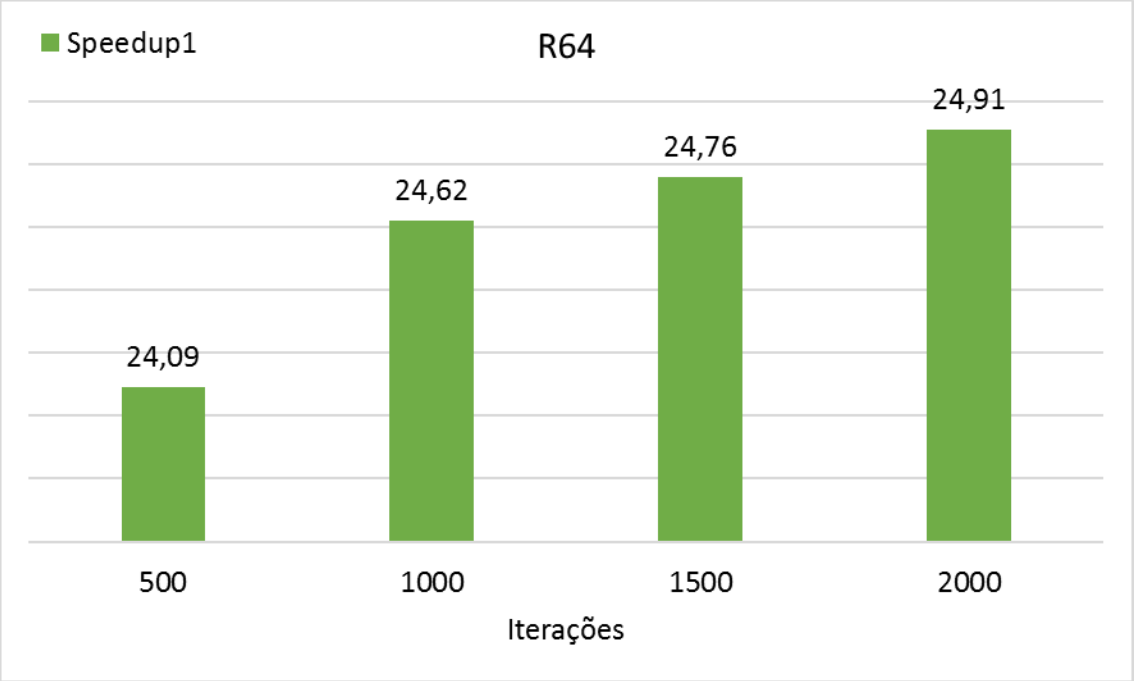

Figura 81 – *Speedup* versus número de iterações (R64)

Por outro lado, como pode ser visto da Figura 82 à Figura 85, o tempo de execução aumenta consideravelmente à medida que o domínio computacional é refinado (para todo o número de iterações investigadas).

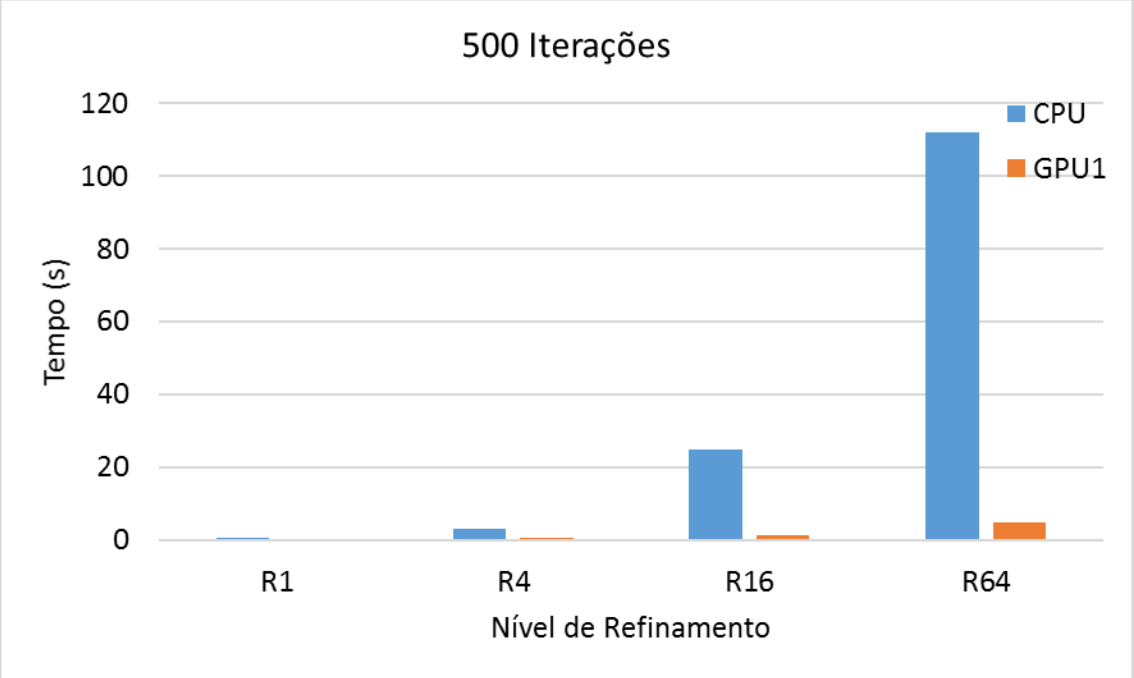

Figura 82 – Tempo de execução versus nível de refinamento (500 iterações)

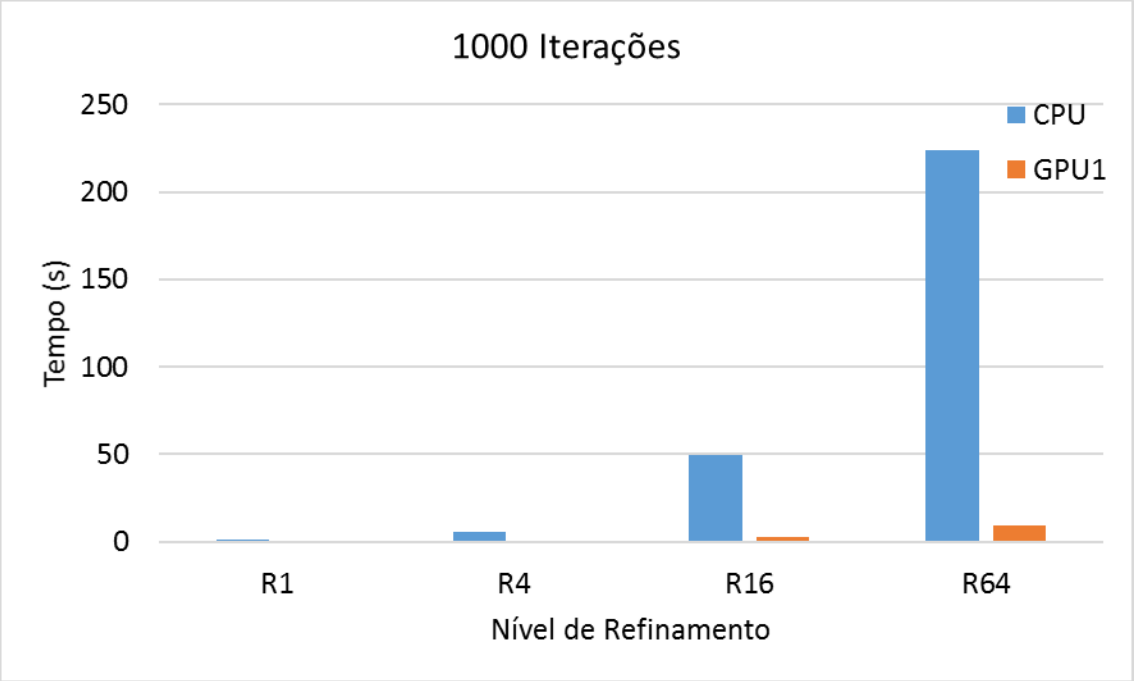

Figura 83 – Tempo de execução versus nível de refinamento (1000 iterações

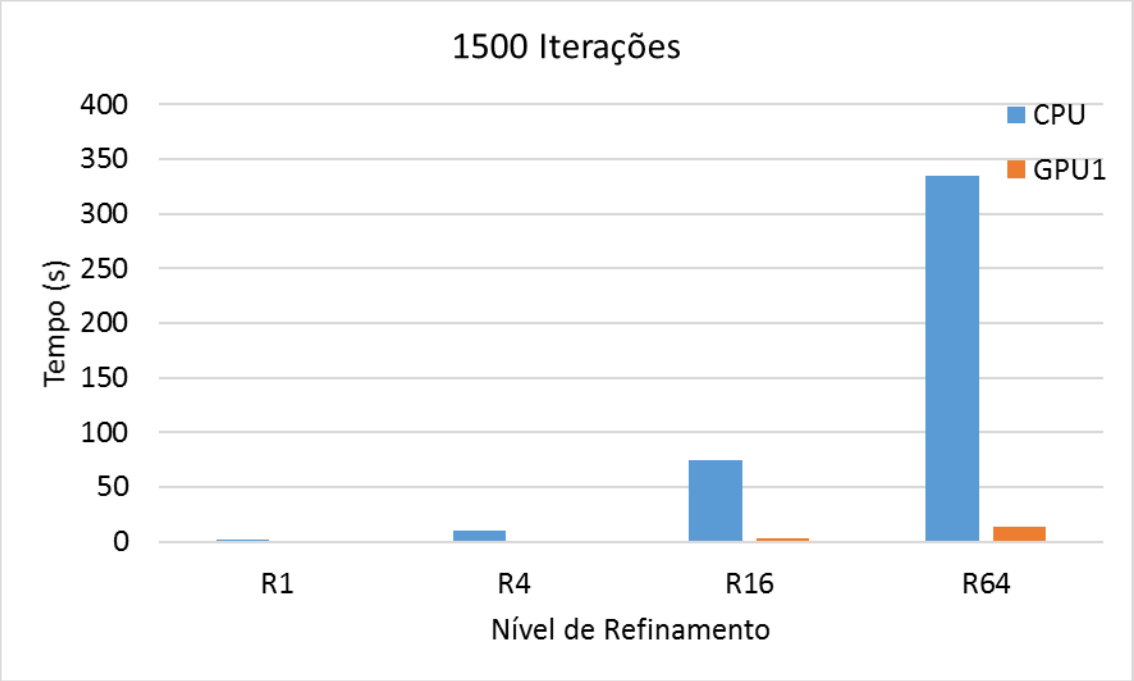

Figura 84 – Tempo de execução versus nível de refinamento (1500 iterações)

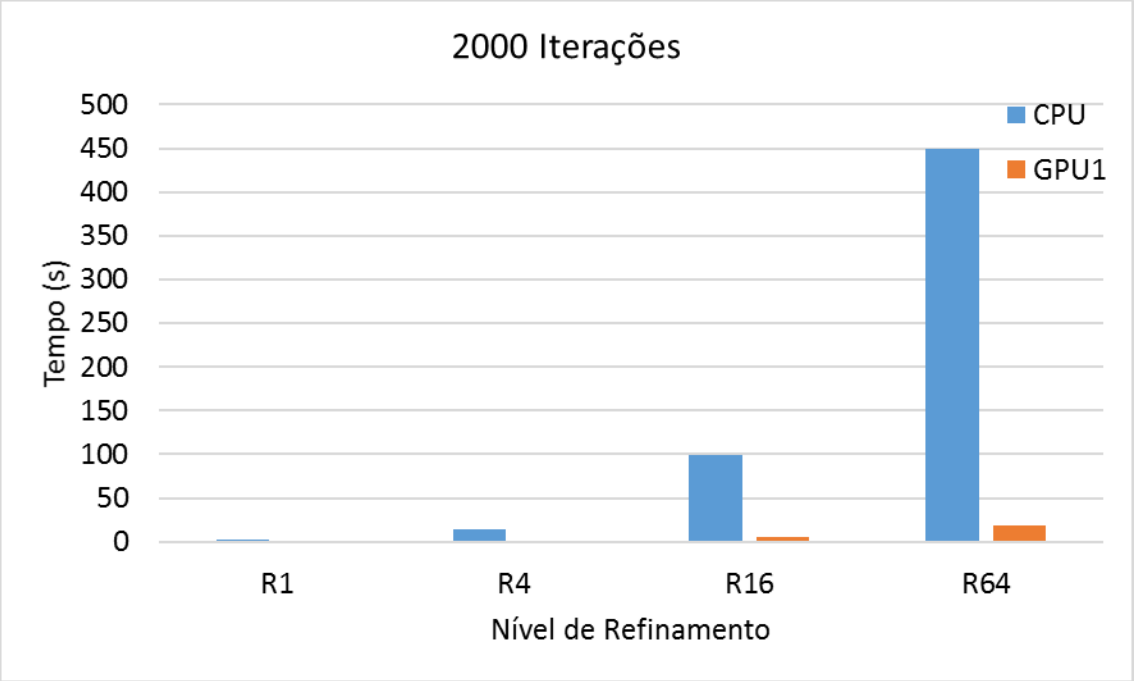

Figura 85 – Tempo de execução versus nível de refinamento (2000 iterações)

Podemos observar da Figura 86 à Figura 89 que as acelerações aumentam significativamente com o refinamento do domínio computacional, no entanto, elas são muito semelhantes para diferentes números de iterações. Apresenta valores pequenos para simulações mais rápidas (R1), um grande gradiente crescente entre R4 e R16, atingindo a marca de cerca de 25 vezes para as simulações com nível de refinamento R64 (para todo o número de iterações). Esse comportamento é esperado devido ao fato de que o tempo gasto no gerenciamento de threads, comunicação de dados entre CPU e GPU e alocação

de memória na GPU tem mais impacto no tempo de execução geral quando o tempo de processamento na GPU é relativamente curto (menores domínios computacionais).

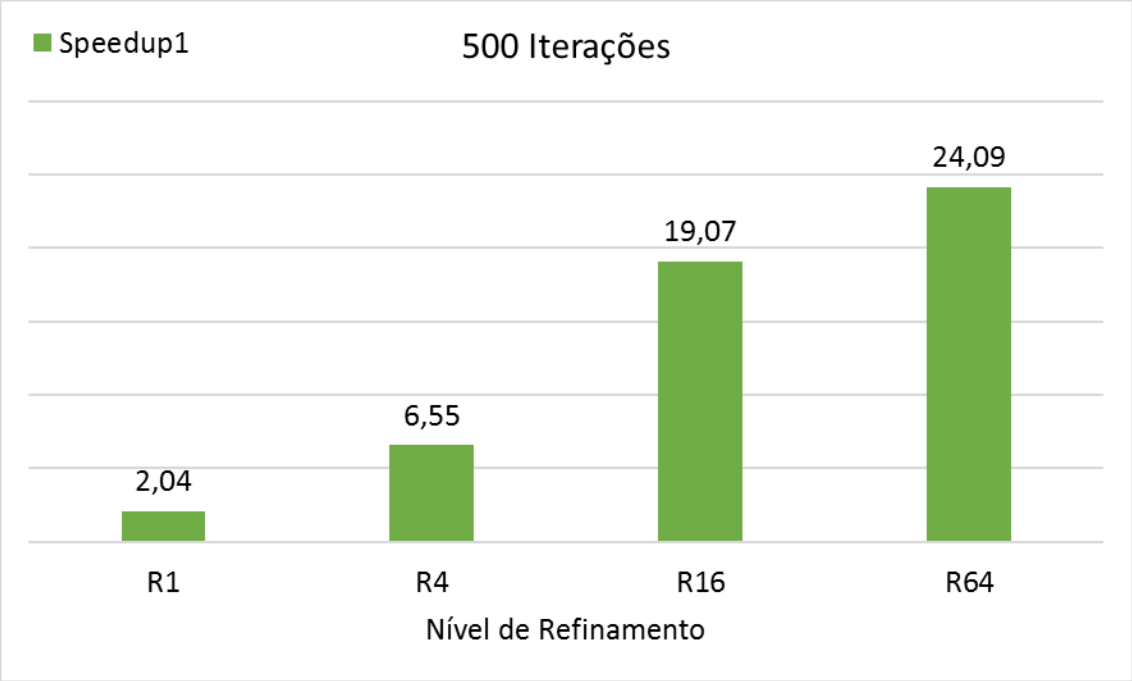

Figura 86 – *Speedups* para diferentes níveis de refinamentos (500 iterações)

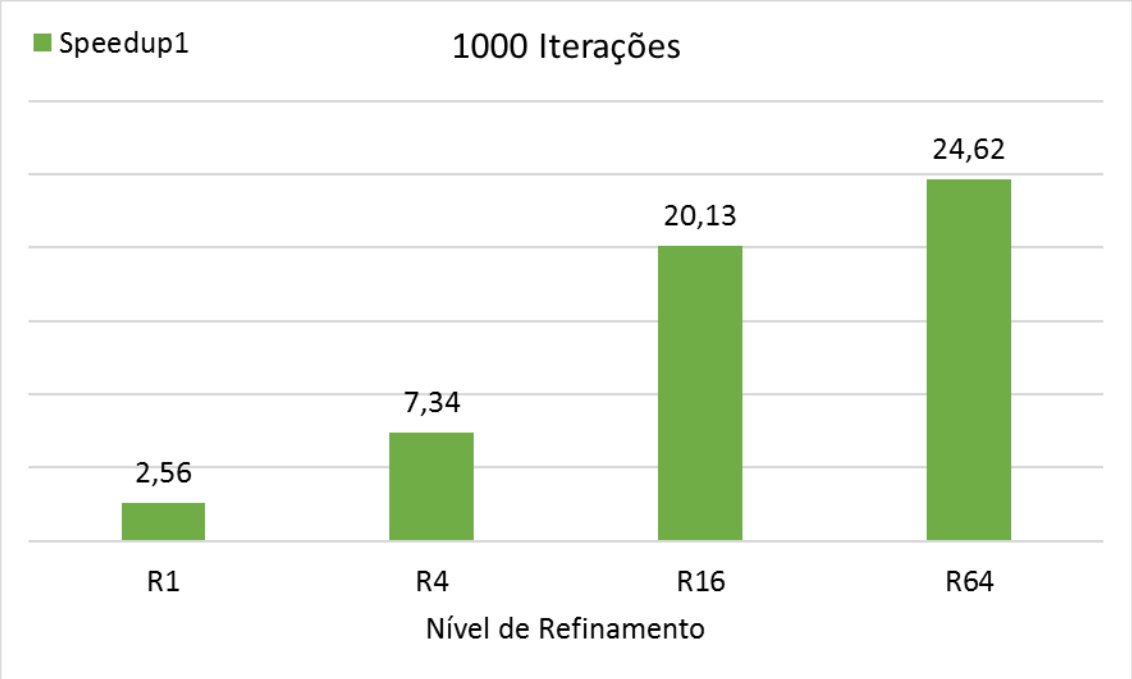

Figura 87 – *Speedups* para diferentes níveis de refinamentos (1000 iterações)

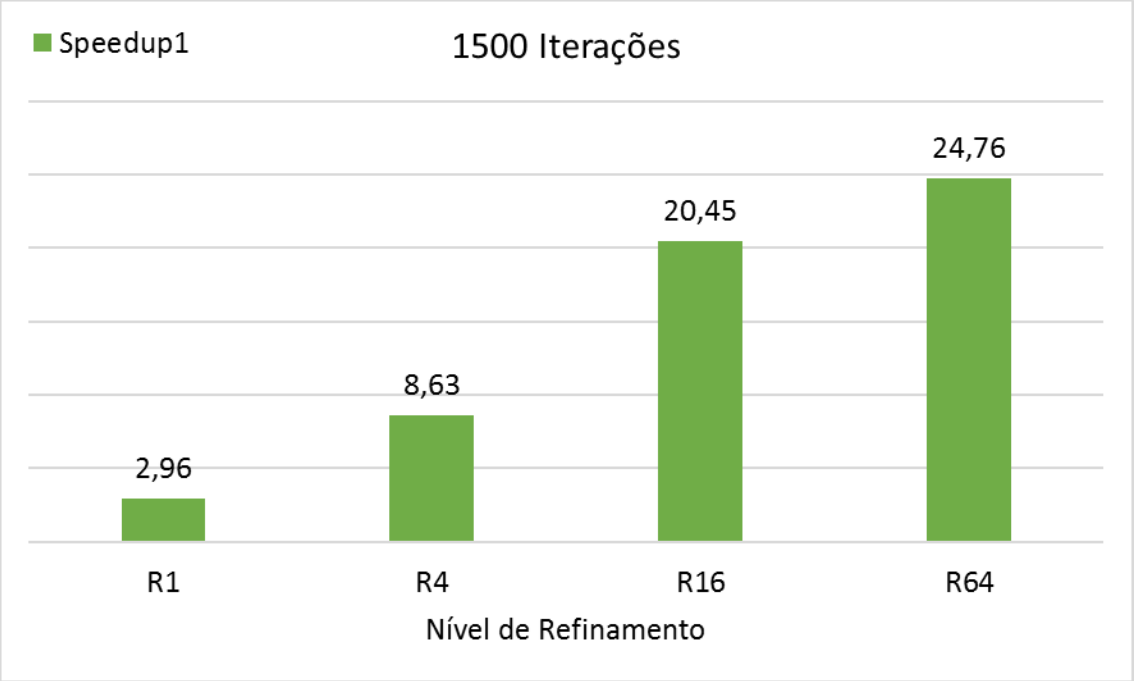

Figura 88 – *Speedups* para diferentes níveis de refinamentos (1500 iterações)

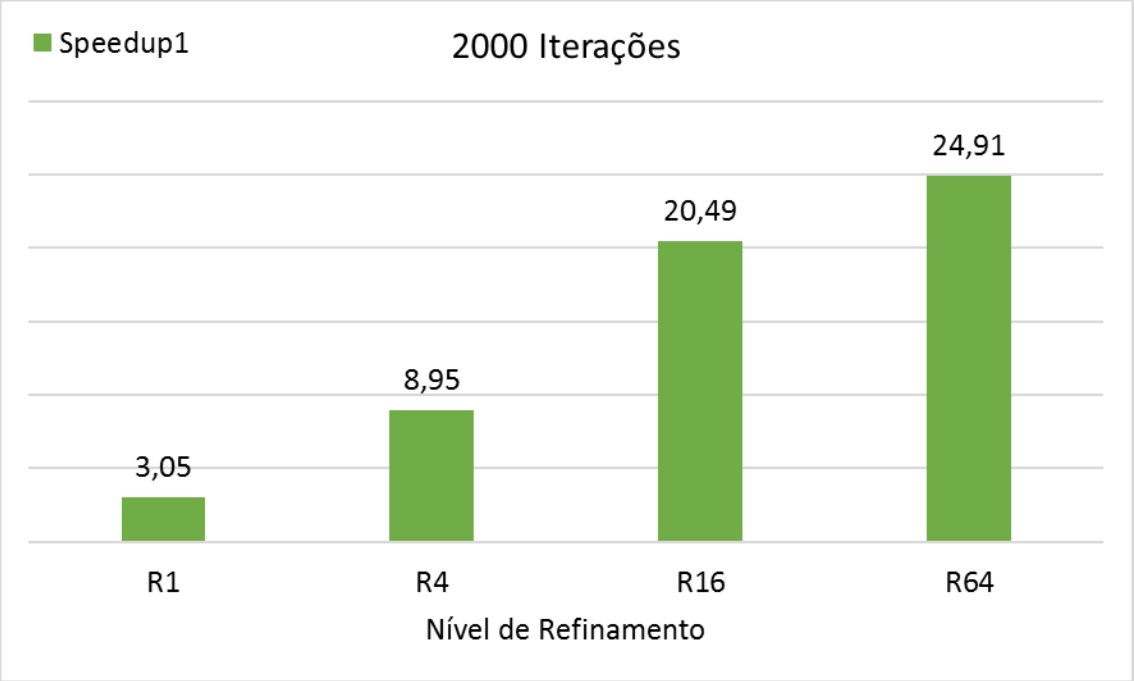

Figura 89 – *Speedups* para diferentes níveis de refinamentos (2000 iterações)

4.0. OTIMIZAÇÃO DO MODELO PARALELO

4.1. OTIMIZAÇÃO DO MODELO PARALELO COM A REALOCAÇÃO DOS PROCESSADORES OCIOSOS

4.1.1. Versão paralela/CUDA do algoritmo 3D-Red-Black com a realocação dos processadores ociosos

Depois de verificado que a abordagem paralela funcionou adequadamente, o desenvolvimento do algoritmo em sua versão paralela com a realocação dos processadores ociosos foi iniciado. Na versão anterior do algoritmo paralelo, pode-se notar que um grande grid de threads é alocado para processar a partição inteira (partição Red ou Black) contudo, devido a utilização do método 3D-Red-Black, metade destes threads passam a ficar ociosos (na execução Red os threads que contém as células Black ficam ociosos e vice-versa) o que não chega a ser um problema para domínios computacionais pequenos e que possam ser alocados nos threads disponíveis na GPU pois, todas as células são executas em um único ciclo paralelo (mesmo que com metade delas ociosa).

Entretanto, quando o domínio computacional aumenta, não existe quantidade suficiente de threads disponíveis na GPU para executar todas as células do domínio computacional em um único ciclo paralelo (cada célula sendo executada por um thread) e devido a este fato, a técnica grid-stride loop é utilizada. De uma forma simplificada, esta técnica, cria dentro de cada thread uma fila de execução onde as células do domínio computacional que não puderam ser alocadas devido a quantidade de threads disponíveis, façam parte desta fila e sejam executadas nos loops posteriores.

Com o aumento do domínio computacional e com a utilização da técnica grid-stride loop passa a ter metade dos loops ociosos, contribuindo assim para o aumento de tempo gasto na execução do algoritmo. Um melhor gerenciamento do uso da memória foi então implementado de maneira que os threads sejam alocados apenas com as células Red, no caso da execução Red ou apenas com as células Black, no caso da execução Black, de forma que não mais tenhamos threads ociosos. Na Figura 90 é mostrado a implementação do kernel CUDA da função Minimizacao_da_ Divergencia_3D_Red_Black_Idle_Kernel.

```
__global__ void Minimizacao_da_ Divergencia_3D_Red_Black_Idle_Kernel (parametros){
 //Declarações e inicializações
 ...
 // Calcula os índices da matriz usando a identificação do thread com Grid-Stride Loops
 for (int K = blockIdx.z * blockDim.z + threadIdx.z; K < NZ; K += blockDim.z * gridDim.z){
  for (int J = blockIdx.x * blockDim.x + threadIdx.x; J < NY; J += blockDim.x * gridDim.x){
   for (int I = blockIdx.y * blockDim.y + threadIdx.y; I < NX; I += blockDim.y * gridDim.y){
    //Otimização do uso dos threads
    KK = K * 2;
    if (Black)
     if ((I + J) % 2 != 0)
      KK--;
    else
     if ((I + J) % 2 == 0)
      KK--;
     IP1 = I + 1;
     ...
     // Calcula os índices em um grande vetor (matrizes 3D são transformadas em 1D)
     index_I = (J + I * NY) * NZ + K;
     index_IP1 = (J + IP1 * NY) * NZ + K;
     ...
     // Coeficiente total de transissão
     TOT = 1.0 / (TX[index_IP1] + TX[index_I]) / DX2 +
                 (TY[index_JP1] + TY[index_J]) / DY2 +
                 (TZ[index_KP1] + TZ[index_K]) / DZ2;
     // Divergente
     D = (VENTOU[index_IP1] - VENTOU[index_I]) / DX +
         (VENTOV[index_JP1] - VENTOV[index_J]) / DY +
         (VENTOW[index_KP1] - VENTOW[index_K]) / DZ;                          // Eq.(4)
     // Diferença da velocidade de perturbação
     delta_Phi = 1.25 * TOT * D;  // 1.25 = Fator de relaxação                 // Eq.(17)
     // Correção da velocidade
     VENTOU[index_I]   = VENTOU[index_I]   + delta_Phi * TX[index_I]   / DX; // Eq.(18)
     VENTOU[index_IP1] = VENTOU[index_IP1] - delta_Phi * TX[index_IP1] / DX; // Eq.(19)

     VENTOV[index_J]   = VENTOV[index_J]   + delta_Phi * TY[index_J]   / DY; // Eq.(20)
     VENTOV[index_JP1] = VENTOV[index_JP1] - delta_Phi * TY[index_JP1] / DY; // Eq.(21)

     VENTOW[index_K]   = VENTOW[index_K]   + delta_Phi * TZ[index_K]   / DZ; // Eq.(22)
     VENTOW[index_KP1] = VENTOW[index_KP1] - delta_Phi * TZ[index_KP1] / DZ; // Eq.(23)
    } // If
   } // I
  } // J
 } // K
```

Figura 90 – Algoritmo paralelo da função Minimizacao_da_ Divergencia_3D_Red_Black_Idle_Kernel

A versão paralela GPU / CUDA 3D-Red-Black da função Minimizacao_da_ Divergencia_3D_Red_Black é a mesma apresentada na Figura 73 pois não sofreu qualquer tipo de alteração.

4.1.2. Consistência da abordagem

Antes da quantificação dos *speedups* e ganhos devido à paralelização, o algoritmo proposto foi aplicado em simulações considerando todos os 20 ventos reais observados e novamente, diversos experimentos foram realizados entre este algoritmo e o algoritmo da primeira implementação. Como resultado de tais experimentos, foram produzidos resultados idênticos ao algoritmo anterior.

4.1.3. Resultados

Considerando que o algoritmo paralelo com a realocação dos processadores ociosos já está consistente, o objetivo desta seção é apenas quantificar e analisar os tempos de execução dos *speedups* obtidos com o uso do programa baseado em GPU com a realocação dos processadores ociosos. Para isso, foi escolhido um único vento observado (Vento #2) e simulações com todos os níveis de refinamento (R1, R4, R16 e R64) e vários números de iterações (500, 1000, 1500 e 2000) foram investigados.

A Tabela 22 mostra os resultados comparativos entre os tempos de execução (em segundos) da rotina WEST do módulo de Campo de Vento (média de 6 execuções) para as implementações sequenciais (CPU), primeira implementação paralela (GPU_1) e para a implementação paralela com a realocação dos processadores ociosos (GPU_2) para os diferentes níveis de refinamento e número de iterações.

Ainda nesta tabela, pode-se observar um aumento do *speedup* desta implementação ($Speedup_2$) em relação ao da primeira implementação ($Speedup_1$), demonstrando assim que a realocação dos processadores ociosos teve um efeito positivo no desempenho do algoritmo. As mesmas características de *hardware* utilizados na primeira implementação paralela, aqui foram também utilizados. Os tempos calculados

para as versões de GPU novamente incluem a execução de kernels na GPU, alocação de memória na GPU e transferência de dados para a GPU.

500					
	CPU	**GPU$_1$***	**GPU$_2$***	**Speedup$_1$**	**Speedup$_2$**
R1	0,52	0,25	0,24	2,04	2,22
R4	3,00	0,46	0,40	6,55	7,44
R16	24,91	1,30	1,12	19,07	22,28
R64	112,24	4,66	4,13	24,09	27,21
1000					
	CPU	**GPU$_1$***	**GPU$_2$***	**Speedup$_1$**	**Speedup$_2$**
R1	1,02	0,40	0,35	2,56	2,94
R4	5,94	0,81	0,66	7,34	8,95
R16	49,74	2,47	2,08	20,13	23,89
R64	223,68	9,09	7,98	24,62	28,02
1500					
	CPU	**GPU$_1$***	**GPU$_2$***	**Speedup$_1$**	**Speedup$_2$**
R1	1,52	0,51	0,49	2,96	3,10
R4	10,02	1,16	0,94	8,63	10,70
R16	74,67	3,65	3,05	20,45	24,48
R64	334,67	13,52	11,86	24,76	28,21
2000					
	CPU	**GPU$_1$***	**GPU$_2$***	**Speedup$_1$**	**Speedup$_2$**
R1	2,02	0,66	0,64	3,05	3,18
R4	13,52	1,51	1,26	8,95	10,75
R16	98,76	4,82	4,01	20,49	24,66
R64	449,29	18,04	15,15	24,91	29,66

*Considerando kernel GPU + alocação de memória na GPU + transferência de dados para a GPU

Tabela 22 - *Speedups* e tempos de execução das implementações sequenciais e paralelas

A Figura 91 até a Figura 94 mostram a influência do número de iterações no tempo de execução para todos os domínios computacionais e para as diferentes implementações do algoritmo.

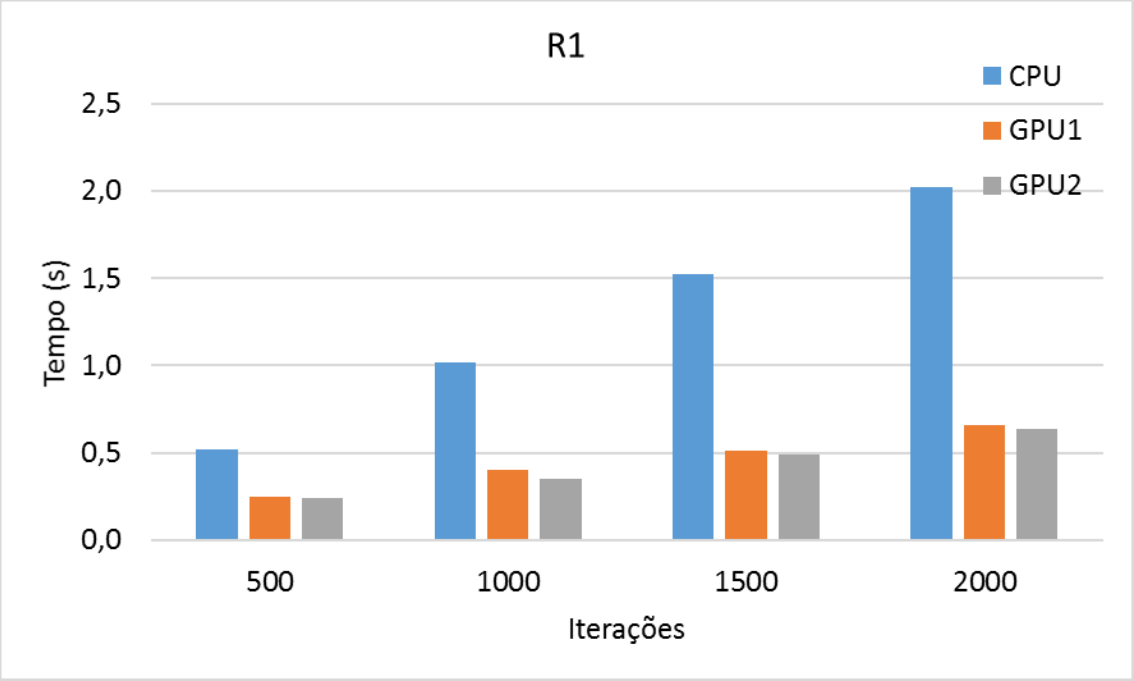

Figura 91 – Tempo de execução versus número de iterações (R1)

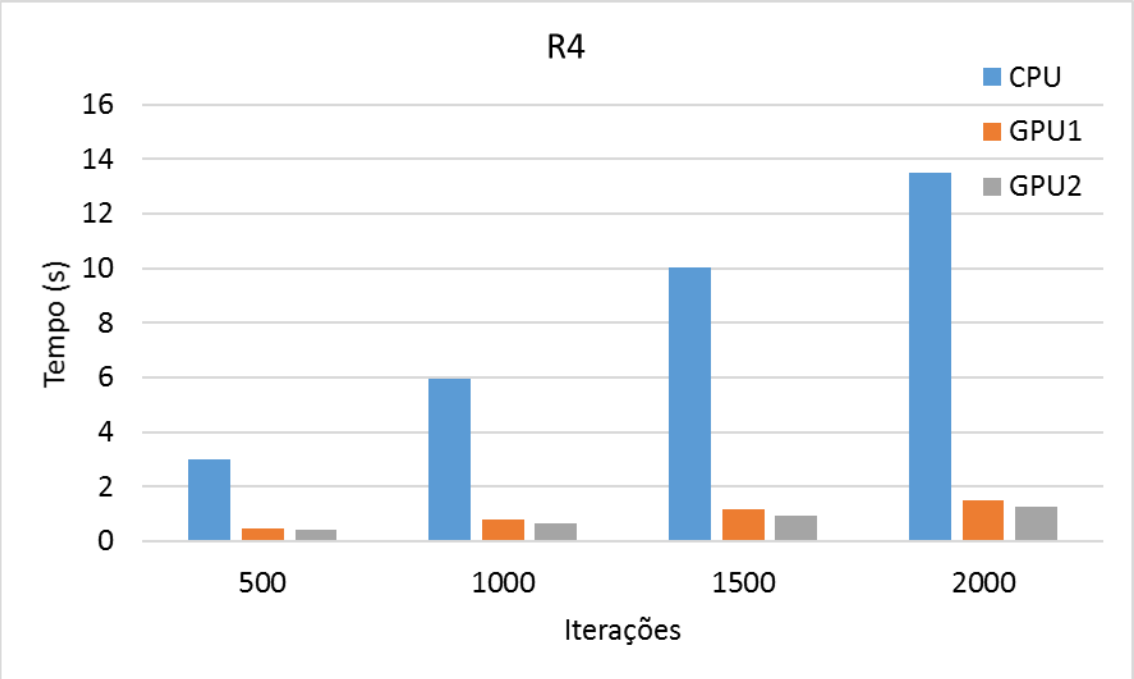

Figura 92 – Tempo de execução versus número de iterações (R4)

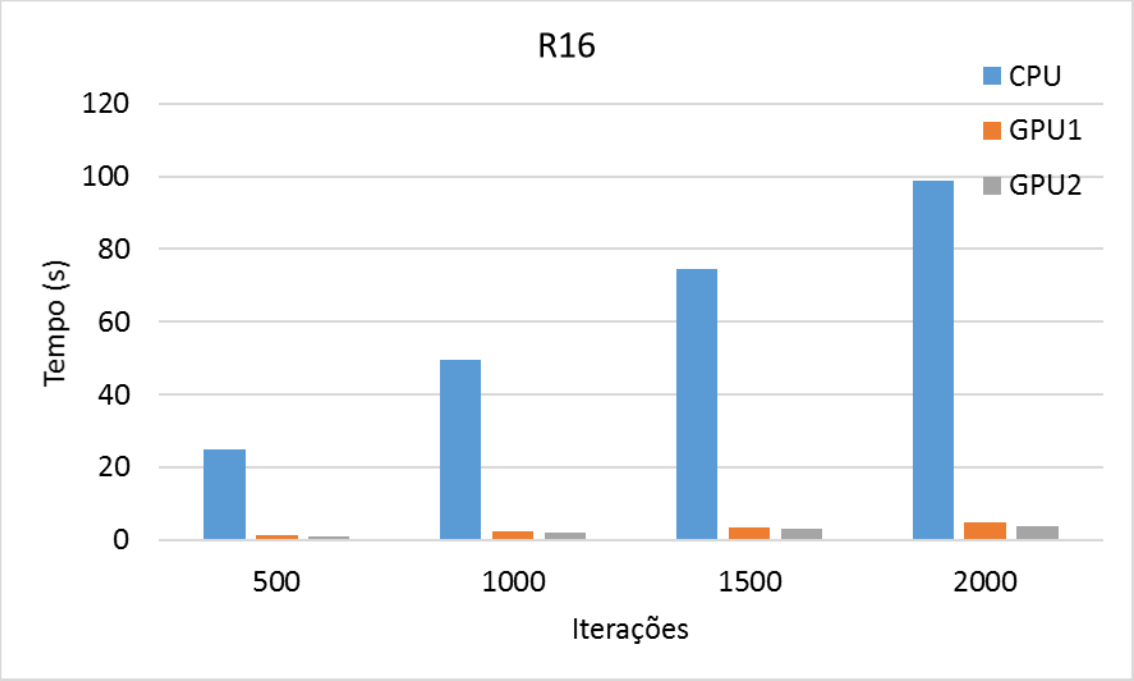

Figura 93 – Tempo de execução versus número de iterações (R16)

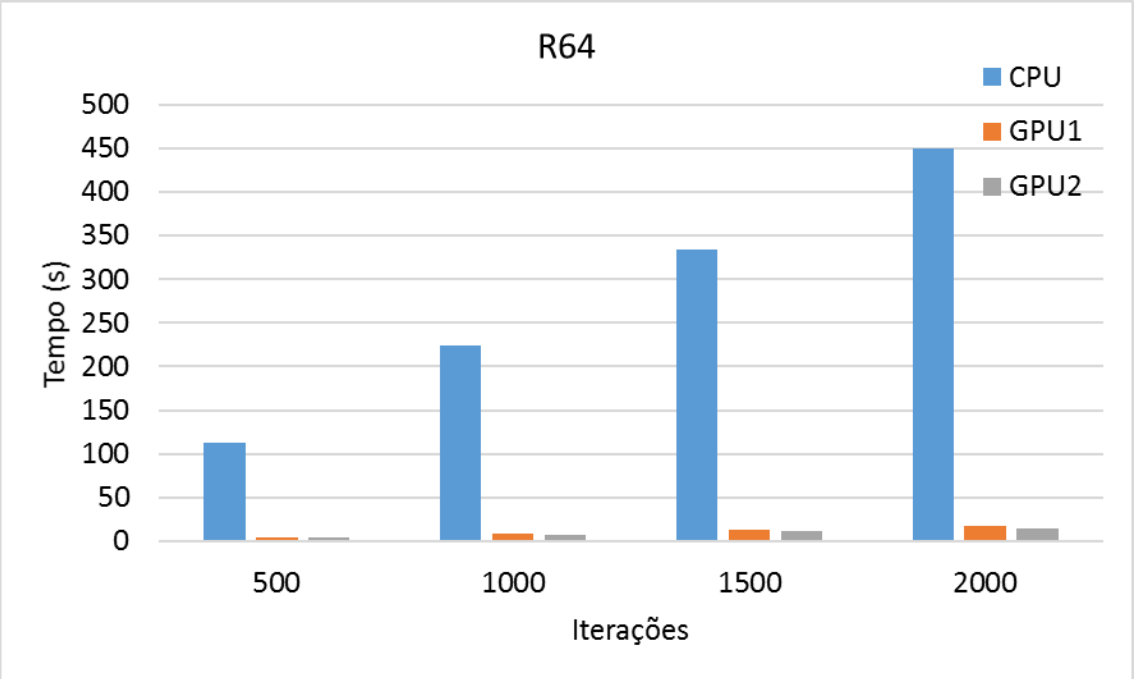

Figura 94 – Tempo de execução versus número de iterações (R64)

Conforme visto, o tempo de execução é aproximadamente proporcional ao número de iterações. Novamente, o *speedup* é significativamente reduzido para

simulações com menos iterações e domínios computacionais menores, onde se enfatiza a contribuição de outras funções no tempo total, aumentando conforme aumenta o número de iterações e o domínio computacional. Tal fato é ilustrado da Figura 95 até a Figura 98, onde pode-se também observar o aumento do *speedup* após a otimização dos processadores ociosos.

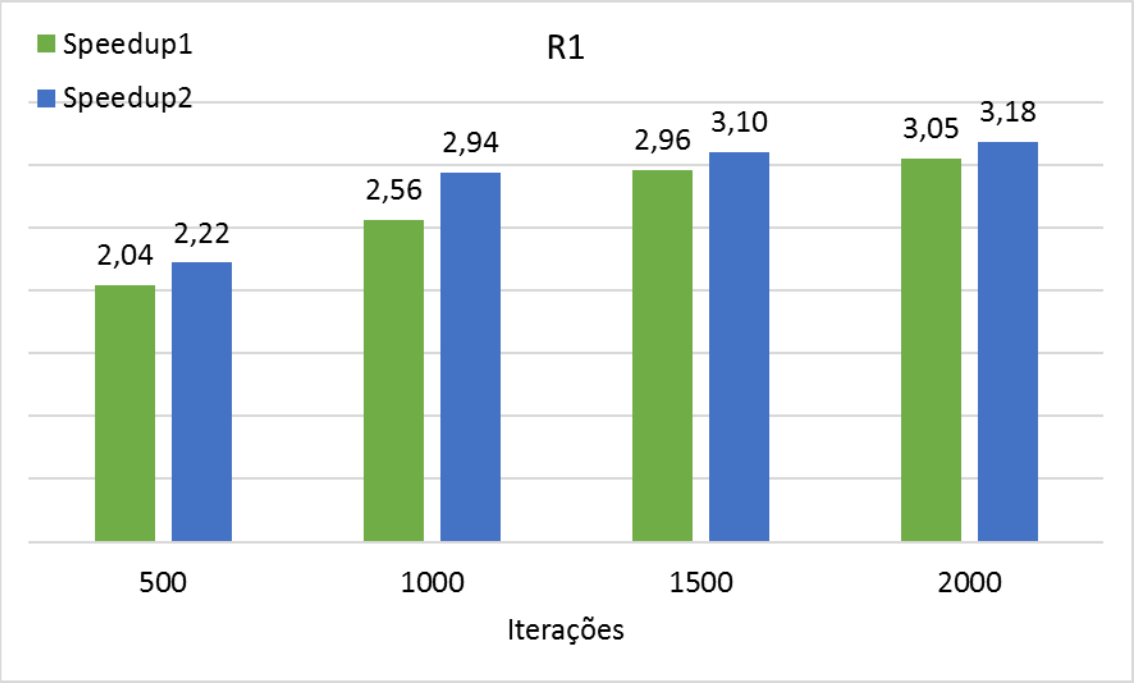

Figura 95 – *Speedup* versus número de iterações (R1)

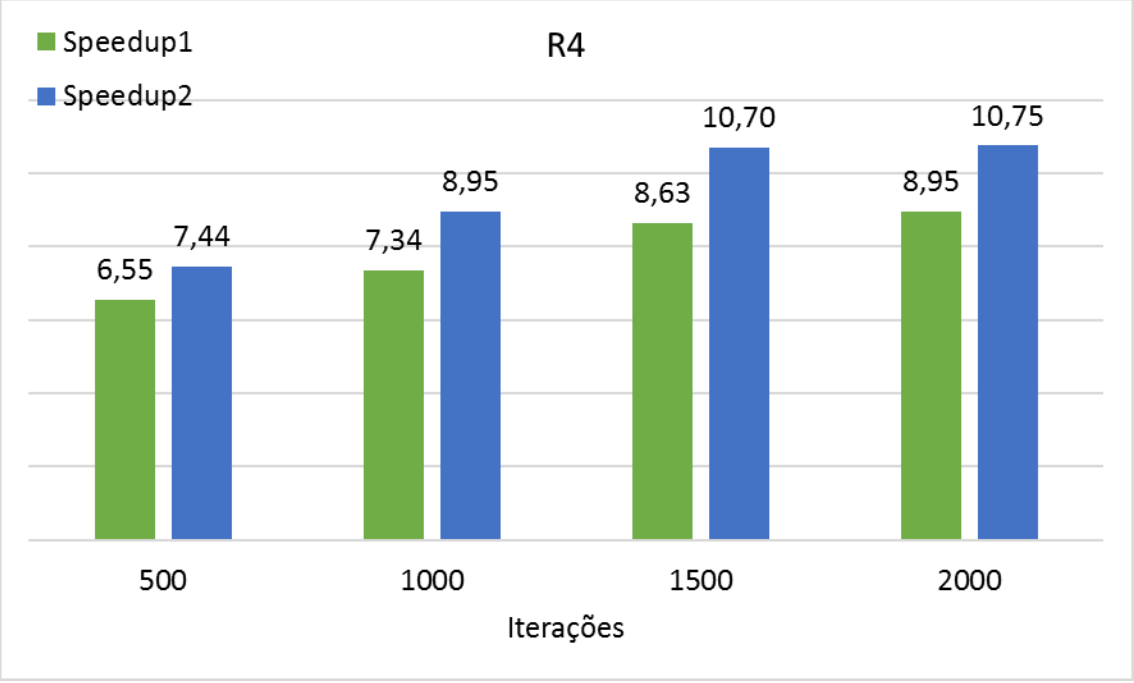

Figura 96 – *Speedup* versus número de iterações (R4)

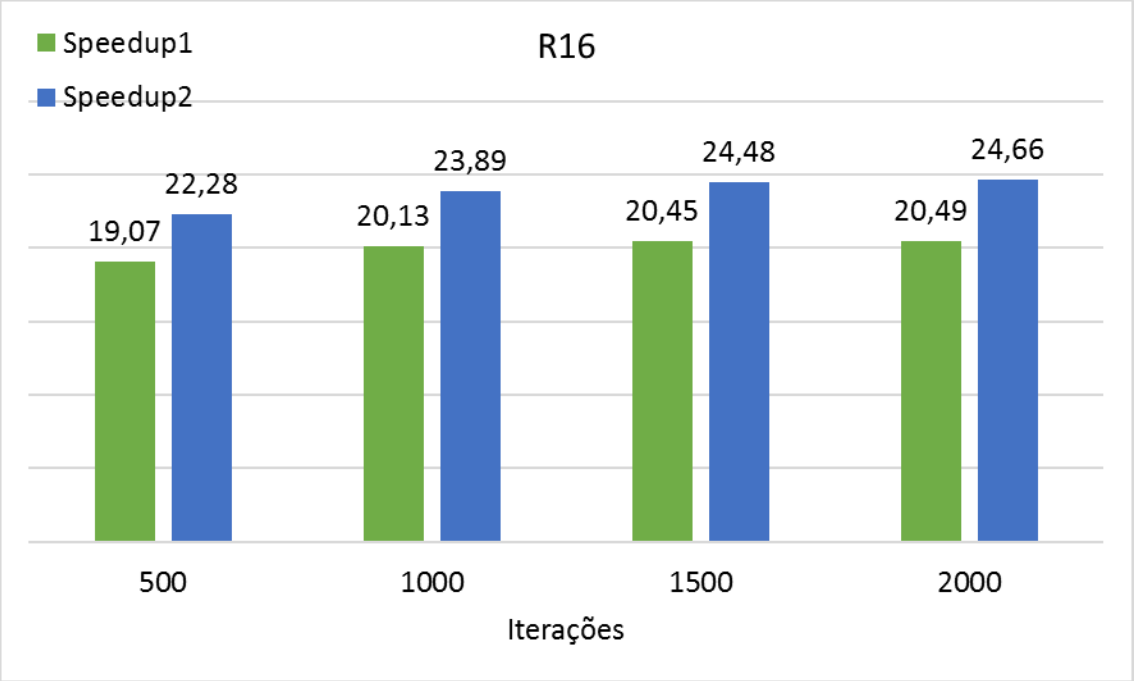

Figura 97 – *Speedup* versus número de iterações (R16)

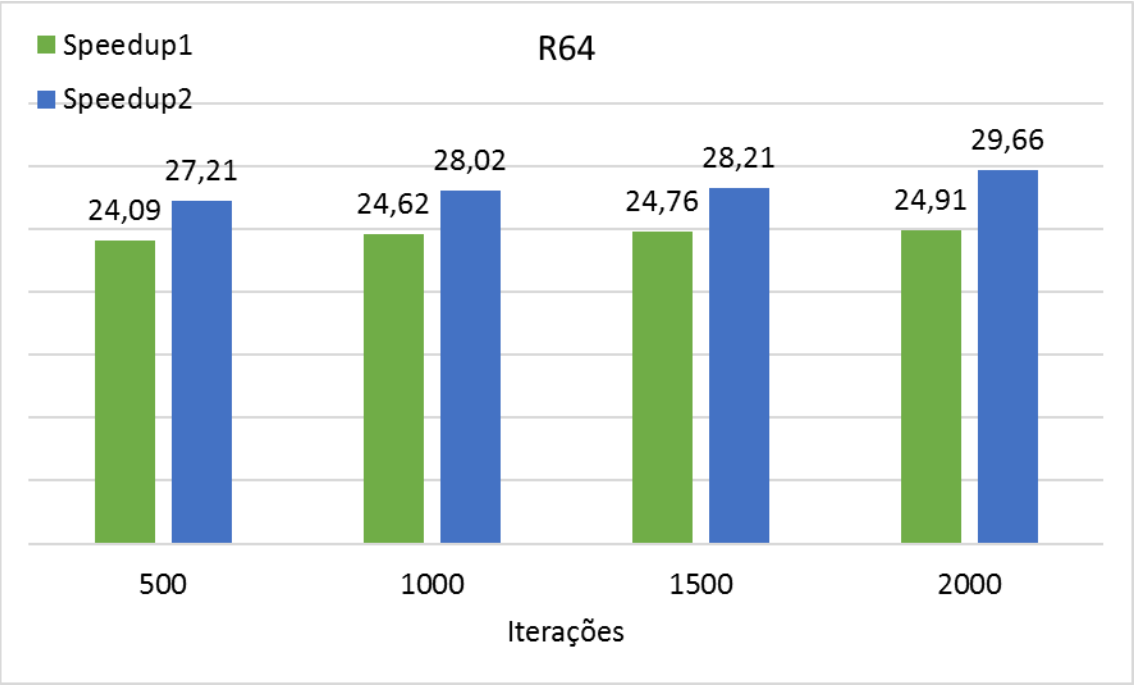

Figura 98 – *Speedup* versus número de iterações (R64)

Pode-se observar, da Figura 99 até a Figura 102, o tempo de execução aumentando à medida que o domínio computacional é refinado (para todo o número de iterações investigadas).

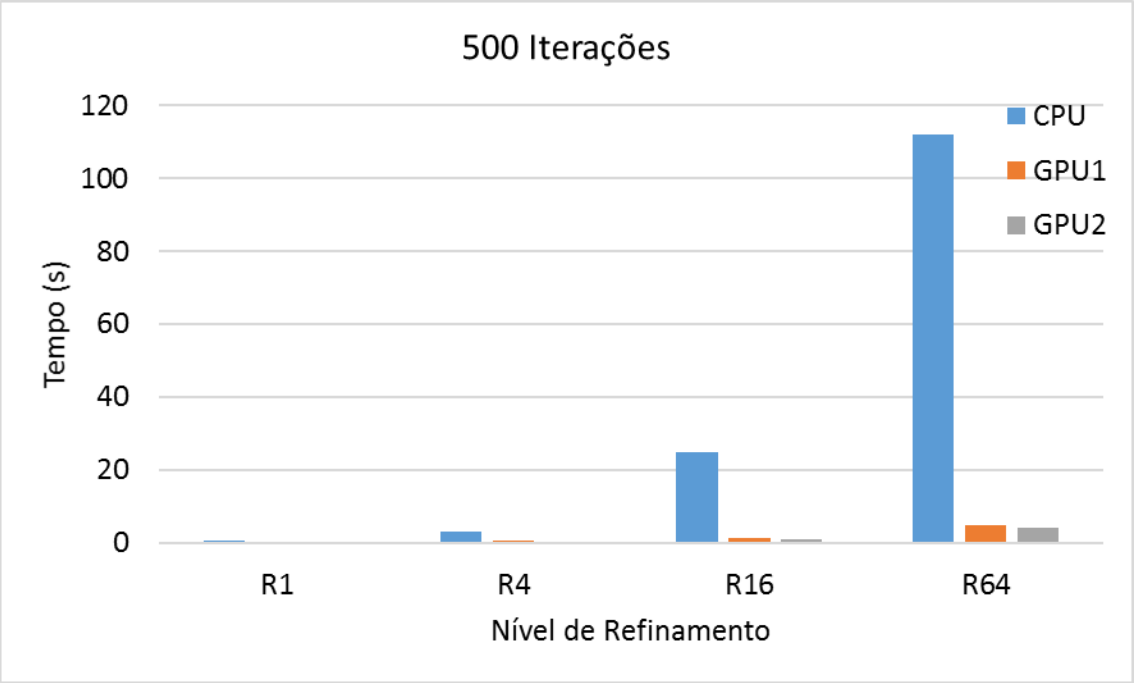

Figura 99 – Tempo de execução versus nível de refinamento (500 Iterações)

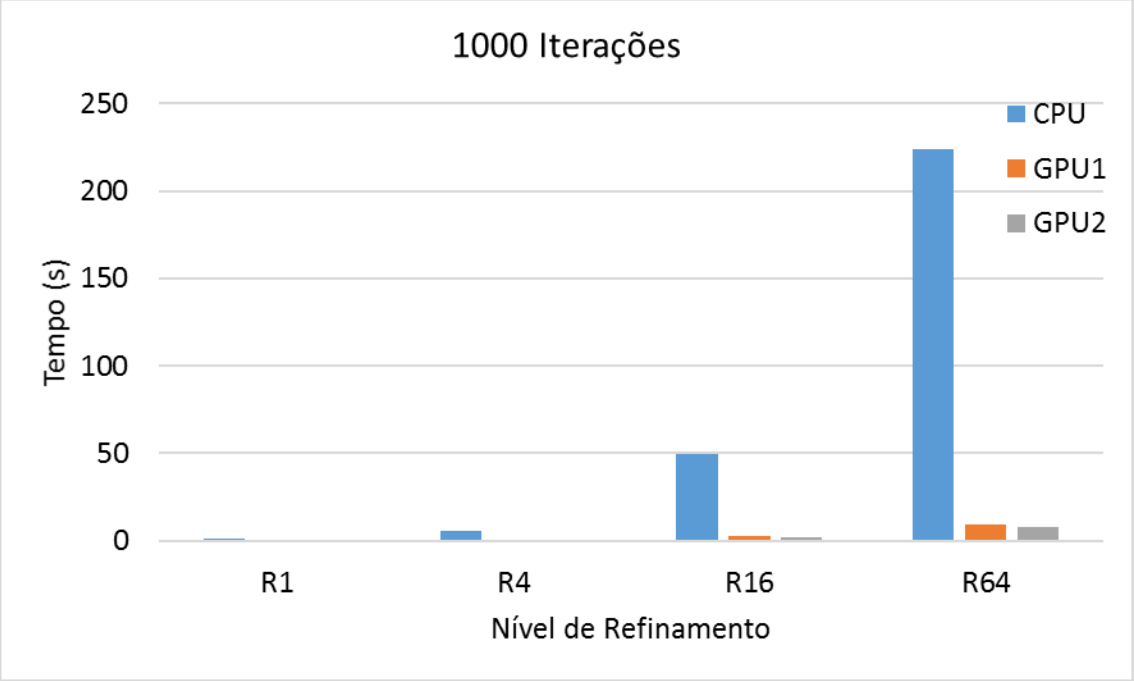

Figura 100 – Tempo de execução versus nível de refinamento (1000 Iterações)

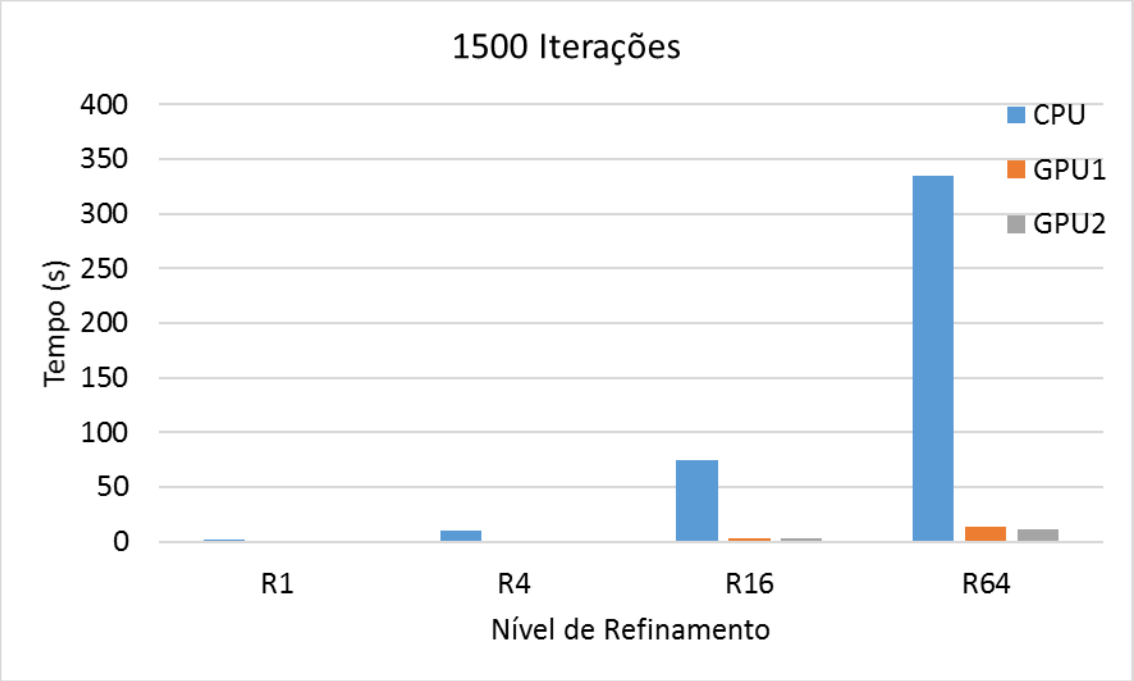

Figura 101 – Tempo de execução versus nível de refinamento (1500 Iterações)

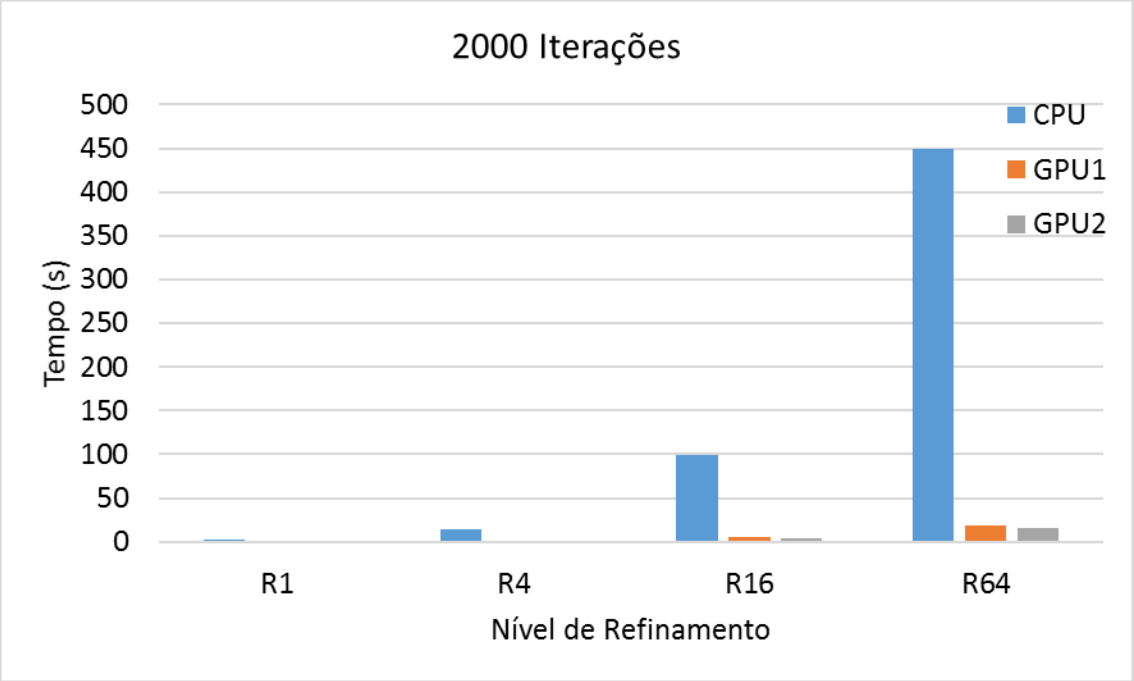

Figura 102 – Tempo de execução versus nível de refinamento (2000 Iterações)

A Figura 103 até a Figura 106 mostram os gráficos comparativos das acelerações com o refinamento do domínio computacional, para todas as diferentes iterações. Todas

as observações feitas para a primeira implementação paralela do algoritmo do Campo de Vento continuam válidas.

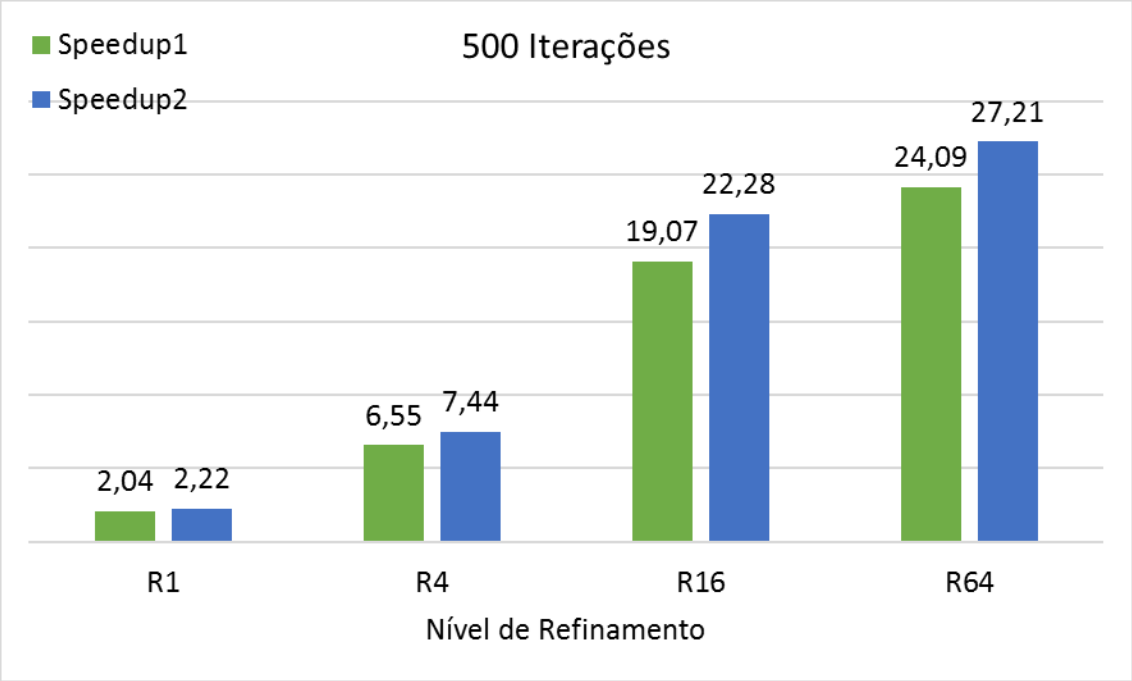

Figura 103 – *Speedups* para diferentes níveis de refinamentos (500 Iterações)

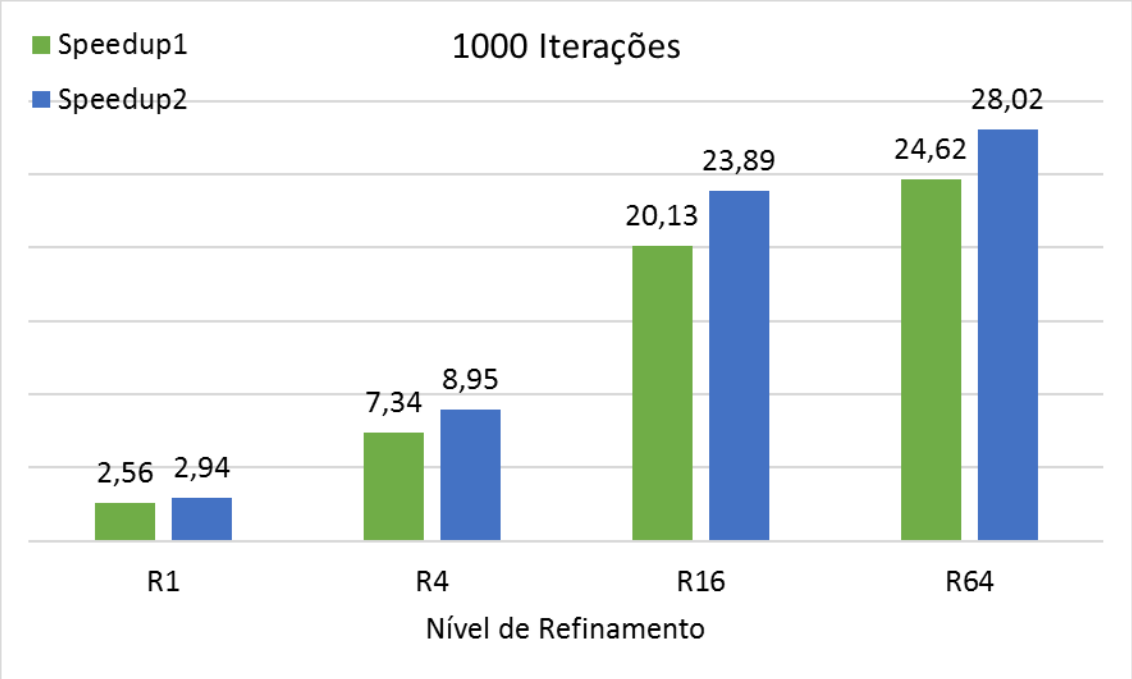

Figura 104 – *Speedups* para diferentes níveis de refinamentos (1000 Iterações)

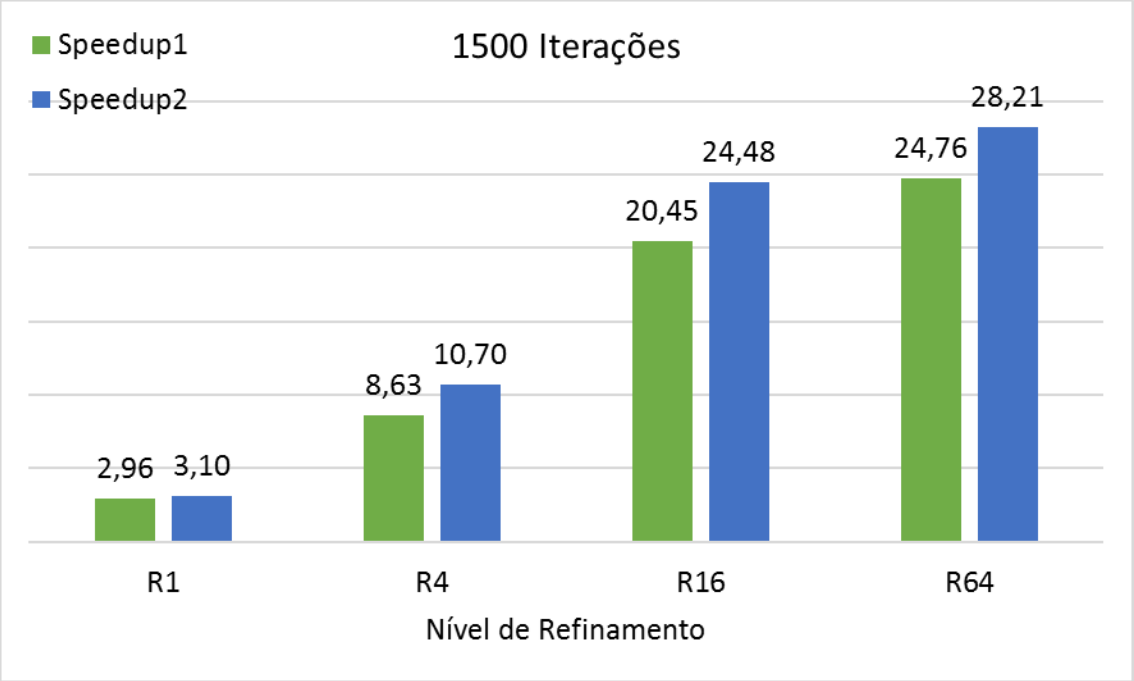

Figura 105 – *Speedups* para diferentes níveis de refinamentos (1500 Iterações)

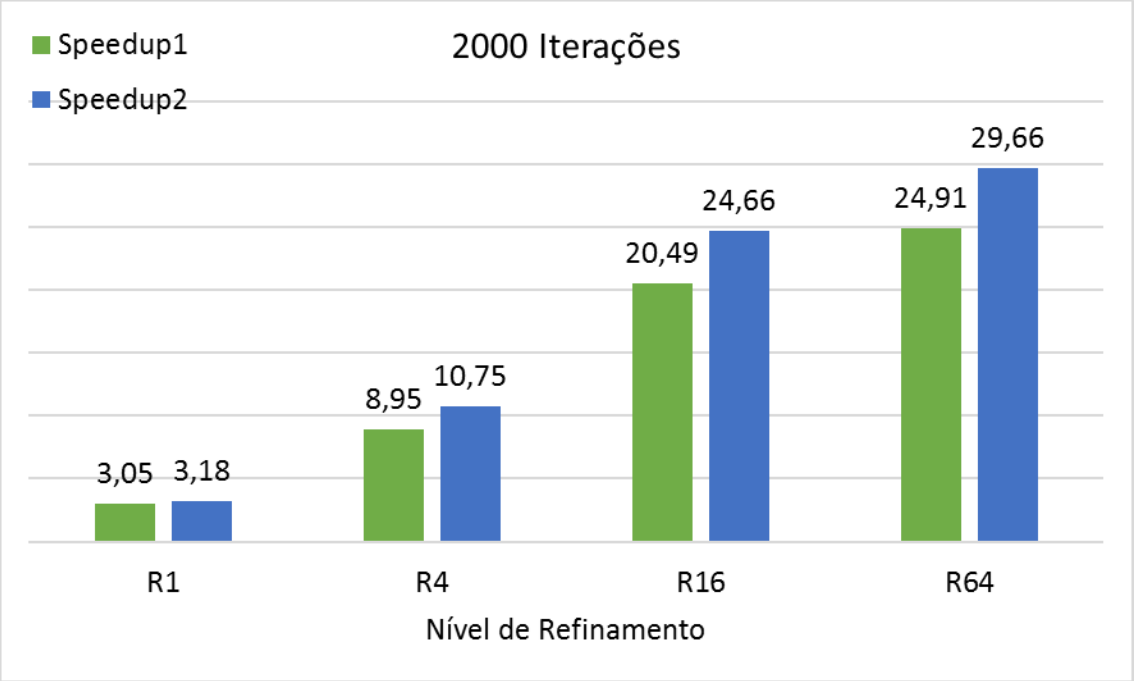

Figura 106 – *Speedups* para diferentes níveis de refinamentos (2000 Iterações)

4.2. OTIMIZAÇÃO DE ALOCAÇÃO DE THREADS VIA PSO

4.2.1. Descrição do problema

Seguindo as recomendações de (WONG, PAPADOPOULOU, *et al.*, 2010) onde os autores indicam a preferência do uso inicial de threads por bloco como múltiplos e submúltiplos de 64, várias experiências computacionais com estes valores foram feitas, e foi adotado como ponto de partida para a dimensão dos blocos de threads, o valor de 8 blocos de threads para as dimensões X, Y e Z. Este valor foi baseado no menor tempo de execução do algoritmo do Campo de Vento. Entretanto, conforme o mesmo autor cita, seu trabalho e as técnicas nele apresentadas foram desenvolvidas para uma GPU Nvidia GT200 (NVIDIA CORPORATION, 2008b) e uma melhor análise em outras arquiteturas de GPU devem ser exploradas.

De uma forma geral, é desejado dimensionar os blocos por threads de forma a coincidir com os dados e simultaneamente maximizar a ocupação dos threads, ou seja, o máximo de threads ativos ao mesmo tempo. Os principais fatores que influenciam a ocupação são o uso de memória compartilhada, o uso de registradores e o tamanho do bloco de threads. Entretanto, antes de se preocupar com o uso da memória compartilhada e com o uso dos registradores, a principal preocupação deve estar no correto dimensionamento dos blocos com base no número máximo de threads e blocos que correspondem à capacidade de computação da placa GPU.

Existem diversas formas de se implementar o dimensionamento correto, por exemplo, em uma GPU onde cada SMX possa ter no máximo 16 blocos e 2048 threads, significa que se for alocado 128 threads por bloco, poderiam caber 16 blocos nos SMX antes de atingir o limite de 2048 threads, mas também poderiam ser alocados 256 threads

com 8 blocos e ainda assim, estar usando todos os segmentos disponíveis com plena ocupação.

Embora (NVIDIA CORPORATION, 2008a) mencione que o uso de múltiplos e submúltiplos de 64 sejam as melhores escolhas para definir o tamanho dos blocos de threads, (NVIDIA CORPORATION, 2008b) desencoraja o uso de tais valores, dizendo que valores maiores seriam melhores. Trabalhos como (RYOO, RODRIGUES, *et al.*, 2008) e (VOLKOV e DEMMEL, 2008) abordam diversas estratégias sobre a otimização do uso das GPUs entretanto, torna-se evidente que o dimensionamento correto do tamanho de threads por bloco varia de acordo com o modelo da GPU e, com o próprio algoritmo em si.

No presente trabalho, devemos ressaltar a grande dificuldade de alocação dos threads por blocos devido a técnica grid-stride loop onde, cada thread é responsável por uma fila de execução, e desta forma não sendo trivial a divisão do domínio computacional entre os diversos threads disponíveis de forma, que todos tenham a mesma quantidade de itens na fila com o maior número de threads alocados. Com o aumento do domínio computacional, simplesmente não basta alocar todos os threads disponíveis da GPU e sim, que se tenha filas pequenas com todos os threads terminando suas execuções no mesmo ciclo.

Desta forma, neste trabalho, foi aplicada uma otimização por enxame de partículas (Particle Swarm Optimization ou PSO) (KENNEDY e EBERHART, 1995) para iterativamente encontrar uma solução candidata para o correto tamanho de threads por bloco que minimize o tempo do algoritmo do Campo de Vento. Sendo mais especifico, o problema de otimização destes parâmetros é um problema de minimizar uma função objetivo, definida pelo par (S, f), no qual:

S é a região factível, é um conjunto finito de todas as possíveis soluções para o problema tratado e;

f é a função objetivo que se deseja minimizar.

Este problema de minimização, consiste em determinar uma solução $s^* \in S$, chamada de solução ótima, que atenda a seguinte relação $f(s^*) \leq f(s), \forall s \in S$.

Neste trabalho, a solução ótima, s^*, é o conjunto composto pelos valores x, y, z que correspondem a quantidade de threads por blocos em cada dimensão. Tais valores estão sujeitos as seguintes restrições: a quantidade de threads nas dimensões X e Y não podem ser superiores a 1024, a dimensão Z não ser superior a 64 e a multiplicação das três dimensões não podem exceder 1024. Tais restrições são impostas pela GPU utilizada neste trabalho conforme descrito em (NVIDIA CORPORATION, 2017).

Desta forma, o problema de minimização pode ser escrito da seguinte forma:

Encontre $f(s(x,y,z)^*) \leq f(s(x,y,z)), \forall s \in S \mid x, y, z \in \mathbb{N}^*$, sujeito a:

$$\begin{cases} 1 \leq x \leq 1024 \\ 1 \leq y \leq 1024 \\ 1 \leq z \leq 64 \\ x * y * z \leq 1024 \end{cases}$$

A função *fitness* a ser minimizada foi desenvolvida de tal forma que, se todas as restrições forem satisfeitas, assume o valor do tempo gasto para executar a rotina WEST. Caso contrário, é penalizada ao desacordo sobre a restrição. A descoberta do valor da penalização foi feita através do problema inverso, ou seja, ao invés de tentar minimizar a rotina WEST do Campo de Vento, foi primeiro executada uma maximização desta rotina de forma que se todas as restrições fossem satisfeitas, o resultado da função *fitness* assume o valor do tempo gasto para executar a rotina WEST, caso contrário é penalizada com o

menor tempo possível, ou seja, zero. Uma vez determinado um provável tempo máximo, pode-se estimar corretamente a penalização.

4.2.2. Estimação da função fitness

Como parâmetros de entrada para a maximização da função objetivo, foram utilizadas em todos os experimentos, partículas com 3 dimensões (x, y, z), coeficiente de aceleração da influência individual e coeficiente de aceleração da influência social ambos iguais a 2 ($C_1 = C_2 = 2$), peso inercial sofrendo uma variação linear entre 0,8 (na primeira geração) até 0,2 (na última geração). O número de partículas, a quantidade de gerações e a semente foram escolhidas aleatoriamente para cada experimento e são exibidos juntos com os resultados.

Nos primeiros experimentos, as dimensões X e Y tiveram seus valores limitados entre 1 e 1024 e a dimensão Z teve seu valor limitado entre 1 e 64 (devido às limitações da GPU) entretanto, devido a última restrição ($x * y * z \leq 1024$) na grande maioria dos casos esta restrição não era atendida e o algoritmo penalizado. Diversos experimentos foram realizados com o espaço de busca total (1024, 1024, 64) entretanto, observou-se que o algoritmo de busca na grande maioria das simulações atingia o número máximo de gerações e não obtinha sucesso (média da fitness próxima de zero). Diversos experimentos foram realizados e chegou-se à conclusão que valores entre 1 e 64 para todas as dimensões obtinham melhores resultados. Desta forma, o espaço de busca foi reduzido e para todos os experimentos a seguir, os valores de 1 a 64 passaram a ser utilizados para todas as dimensões.

Foram executados quatro experimentos para a determinação da penalização da *fitness* para o caso da minimização do tempo de execução da rotina WEST. Os parâmetros

utilizados e resultados obtidos nestes experimentos podem ser vistos na Tabela 23. Experimentos adicionais com um número maior de gerações não foram executados pois com a maximização dos tempos, aproximadamente por volta da geração de número 200 cada partícula demorava em torno de 10 a 15 segundos para ser executada, consumindo assim uma enorme quantidade de tempo para o término da geração.

Experimento	Gerações executadas	Semente	População	gBest (ms)	Threads por bloco
#1	1000	123456789	20	20639	(1,11,61)
#2	382	24861793	50	20666	(1,11,59)
#3	239	13579	15	20604	(1,11,59)
#4	105	24680	15	20592	(1,14,62)

Tabela 23 – Parâmetros e resultados da maximização da rotina WEST

A Figura 107, mostra a média dos tempos encontrados pelas partículas em cada geração nos quatro experimentos realizados. Observa-se o deslocamento desta média para o valor de gBest.

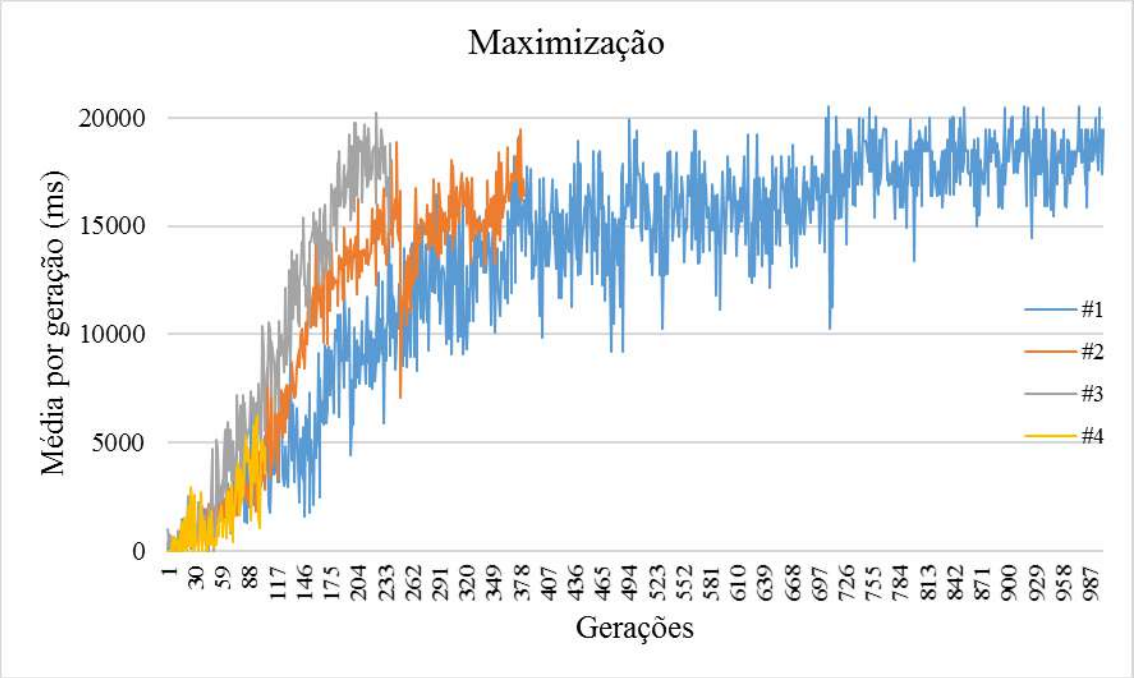

Figura 107 – Gerações versus média por geração (Maximização)

A Figura 108 mostra o gráfico do valor de gBest ao longo das gerações onde pode-se perceber que mesmo no teste 4 onde foi executado o menor número de gerações (105) ainda assim gBest atingiu valores próximos aos outros testes. Pode-se então assim dizer que o algoritmo de maximização cumpriu seu objetivo.

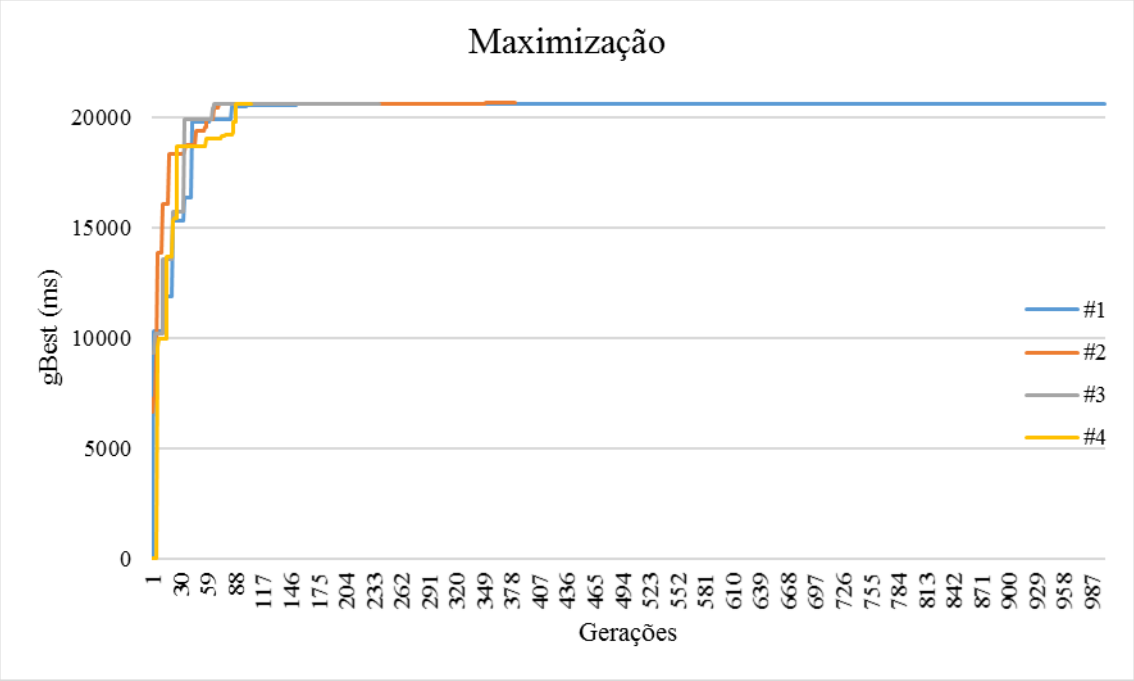

Figura 108 – Gerações versus gBest (Maximização)

A Figura 109 exibe um histograma típico das 10 combinações de threads por blocos (X, Y, Z) mais encontradas nos testes. Em todos os testes realizados, nota-se uma diferença de quase 50% da quantidade da combinação mais encontrada para a segunda.

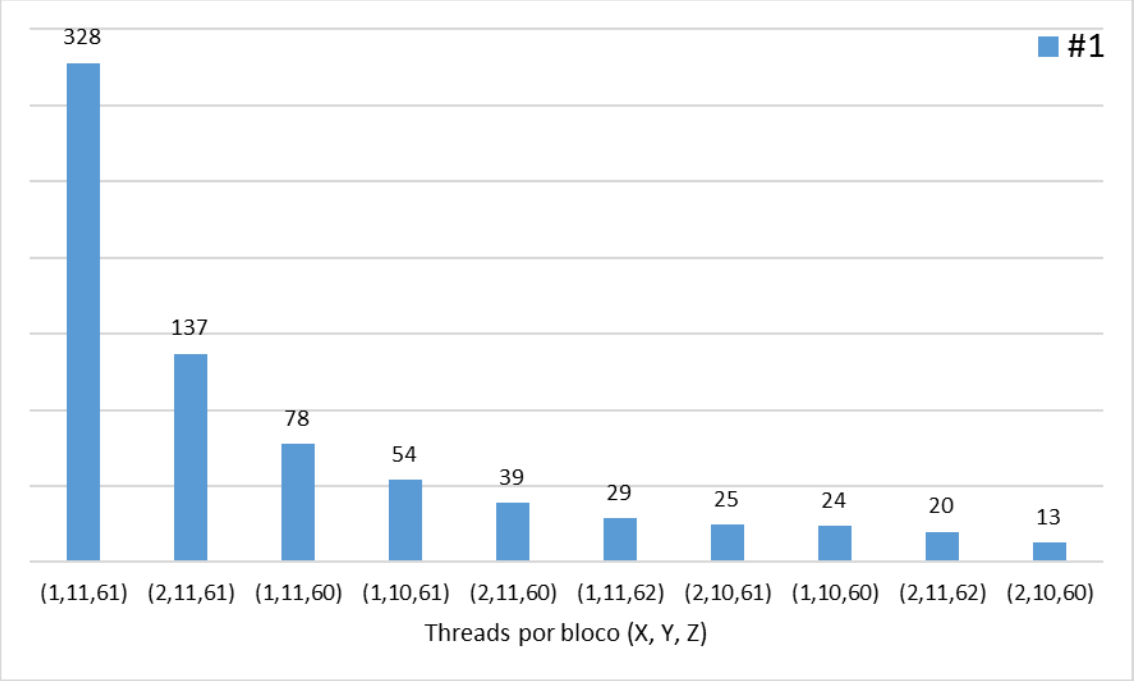

Figura 109 – Histograma das 10 combinações mais encontradas (Maximização)

Acredita-se, que se fossem executados mais experimentos, com gerações maiores e com diferentes populações, valores maiores de gBest poderiam ser encontrados, entretanto, tal abordagem estaria fora do escopo deste trabalho. Após a obtenção dos resultados da maximização da rotina WEST pelo PSO, pode-se constatar que valores na ordem de 20.000ms podem ser considerados de uma forma geral para o algoritmo paralelo da função Minimizacao_da_ Divergencia_3D_Red_Black_Idle_Kernel como um valor de tempo máximo atingido pela sub-rotina WEST do Campo de Vento. Desta forma, foi estimada como razoável para penalização da *fitness*, um valor de 2,5 vezes o tempo máximo desta rotina maximizada, ou seja, 50.000ms.

4.2.3. Alocação de threads via PSO

Uma vez determinada a penalização da função *fitness*, sete experimentos foram realizados para minimizar o tempo da rotina WEST do Campo de Vento. Como parâmetros de entrada para a minimização da função objetivo, foram utilizadas em todos

os experimentos, partículas com 3 dimensões (x, y, z), coeficiente de aceleração da influência individual e coeficiente de aceleração da influência social ambos iguais a 2 ($C_1 = C_2 = 2$), peso inercial sofrendo uma variação linear entre 0,8 (na primeira geração) até 0,2 (na última geração).

Como parâmetros para o algoritmo, foi escolhido o nível de refinamento R16 executando 2000 iterações. Este nível de refinamento foi escolhido, pois devido ao grande número de gerações executas pelo algoritmo PSO, um nível de refinamento maior iria consumir uma enorme quantidade de tempo. Conforme mencionado antes, o nível de refinamento ideal deverá ser verificado quando todo o SCA estiver convertido, sendo assim, será necessário executar o algoritmo para a alocação de threads novamente com o nível de refinamento desejado.

Cabe ainda ressaltar, que mesmo encontrando a melhor combinação de threads por bloco para o nível de refinamento R16, em experimentos futuros será mostrado que esta combinação ainda assim obtém melhores resultados em outros níveis de refinamento que os valores inicialmente escolhidos para os threads por blocos. O número de partículas, a quantidade de gerações e a semente foram escolhidas aleatoriamente para cada experimento e são exibidos juntos com os resultados na Tabela 27.

Experimento	**Gerações executadas**	**Semente**	**População**	**gBest (ms)**	**Threads por bloco**
#1	1500	123456789	40	1656	(8,1,11)
#2	1500	987654321	10	1670	(8,1,18)
#3	1500	123456789	20	1615	(8,1,14)
#4	1500	13579	30	1650	(8,1,09)
#5	1500	159357	50	1660	(8,1,14)
#6	1000	123456789	20	1640	(8,1,14)
#7	813	159357	50	1640	(8,1,15)

Tabela 24 – Parâmetros e resultados da minimização da rotina WEST

Na Figura 110 vemos o comportamento típico da média da fitness por geração (tempo da rotina WEST) nos experimentos realizados. Pode-se observar que com o avanço das gerações a média por geração da *fitness* tende para gBest, mostrando desta forma que o algoritmo está convergindo.

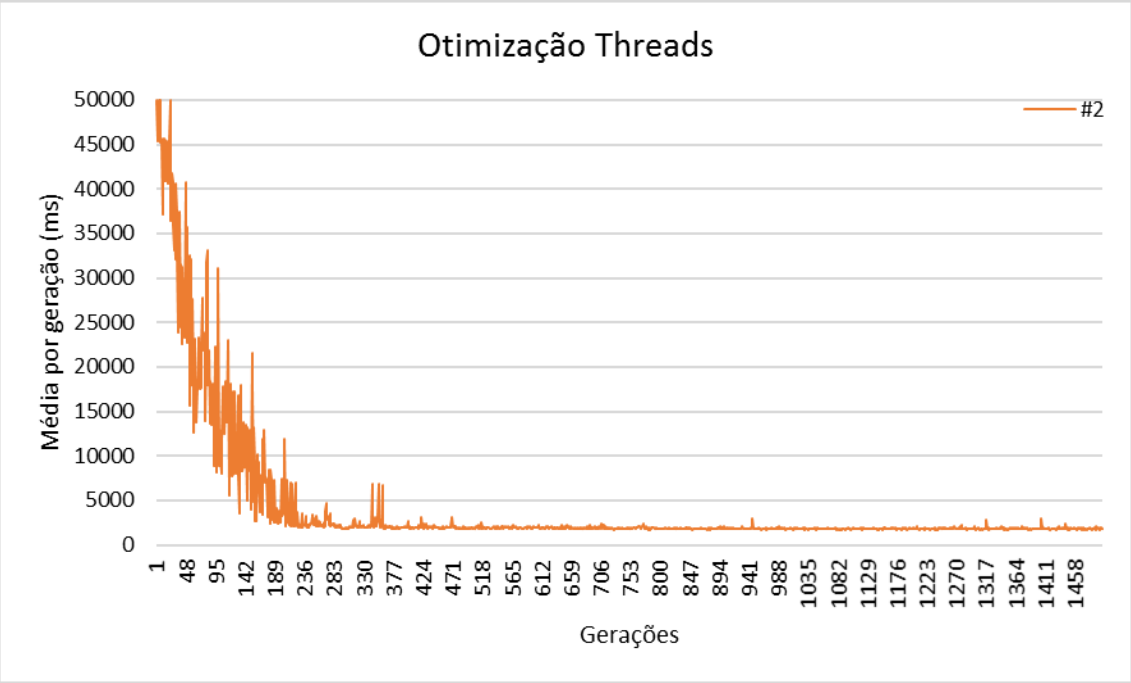

Figura 110 – Geração versus média por geração (Otimização)

Na Figura 111 observa-se a média da fitness por geração de todas as experiências em um único gráfico de forma a observar o comportamento geral.

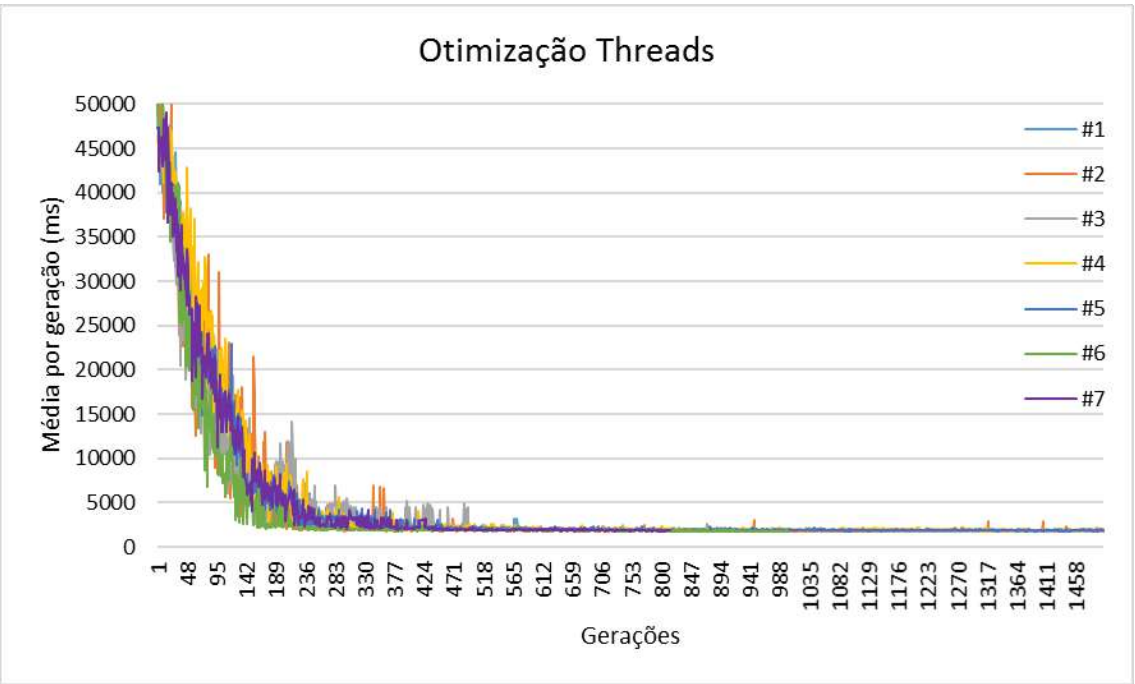

Figura 111 – Geração versus média por geração (Otimização – todos experimentos)

A Figura 112 exibe o comportamento típico do valor de gBest ao longo das gerações entretanto, pode-se constatar que em todas as experiências realizadas, seu valor foi sendo minimizado.

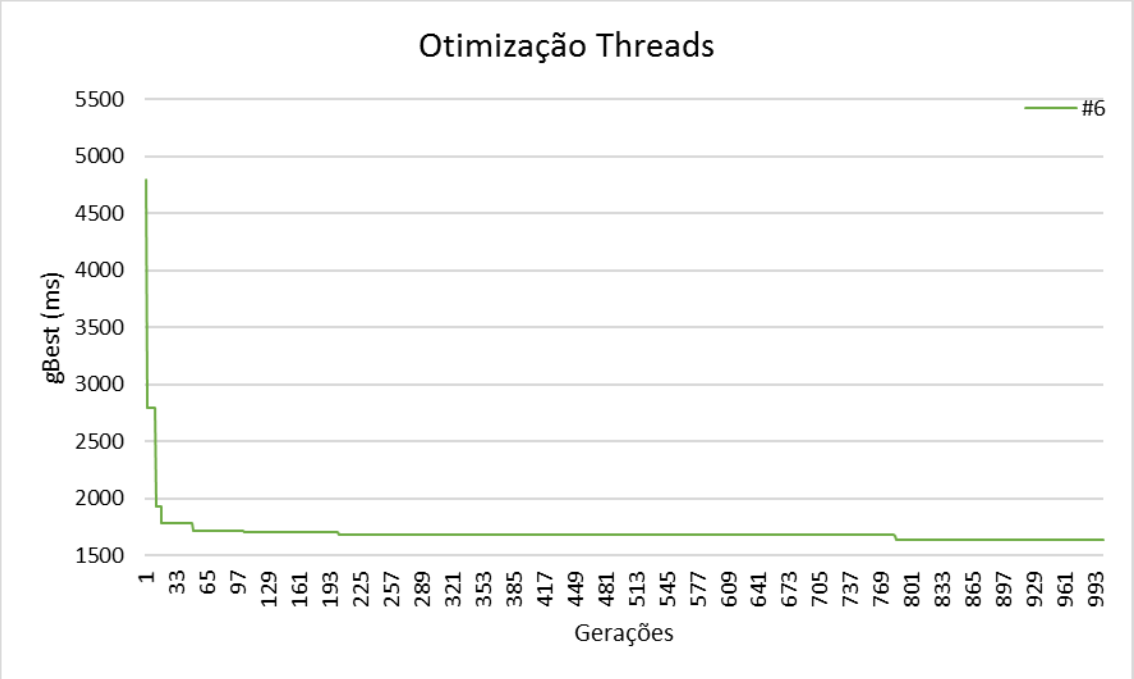

Figura 112 – Gerações versus gBest (Otimização)

A Figura 113 e a Figura 114 mostram o comportamento de gBest ao longo das gerações para todas as experiências realizadas em um único gráfico. Pode-se constatar que mesmo nos experimentos de número 2 e 4 onde gBest demorou um pouco mais para convergir, todos os experimentos obtiveram êxito em minimizar a função fitness ou seja, encontrar uma combinação de threads (X, Y, Z) que minimizasse o tempo de execução da rotina WEST. Desta forma, pode-se dizer que o algoritmo PSO para minimização do tempo de execução da rotina WEST obteve êxito.

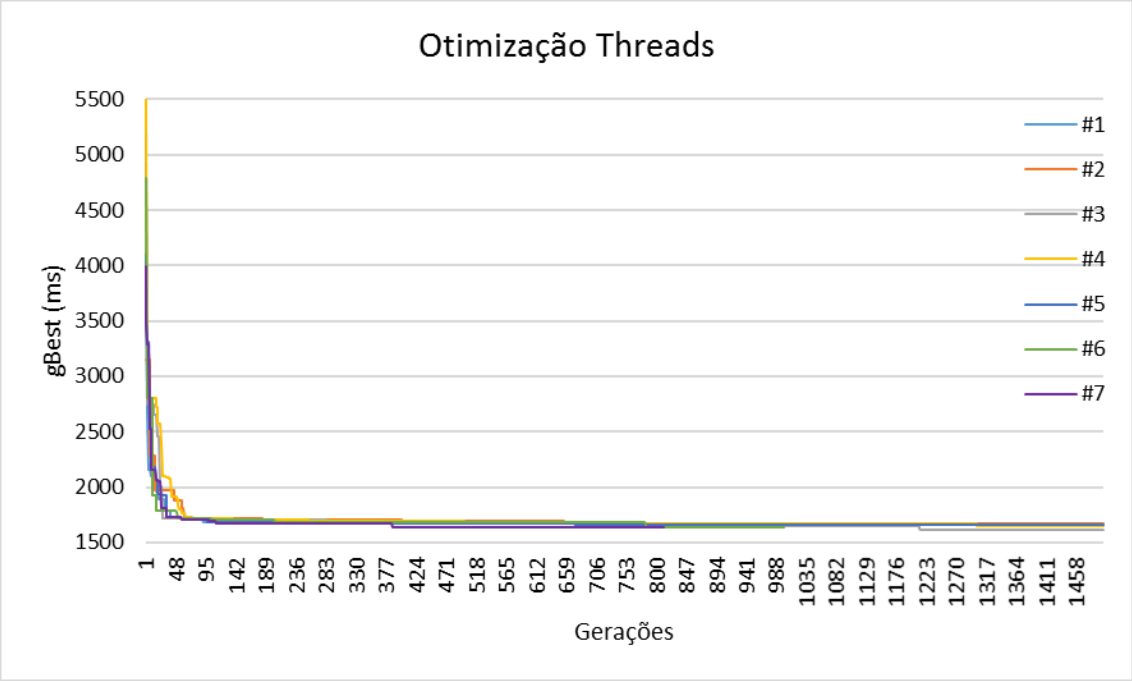

Figura 113 – Gerações versus gBest (Otimização – todos experimentos)

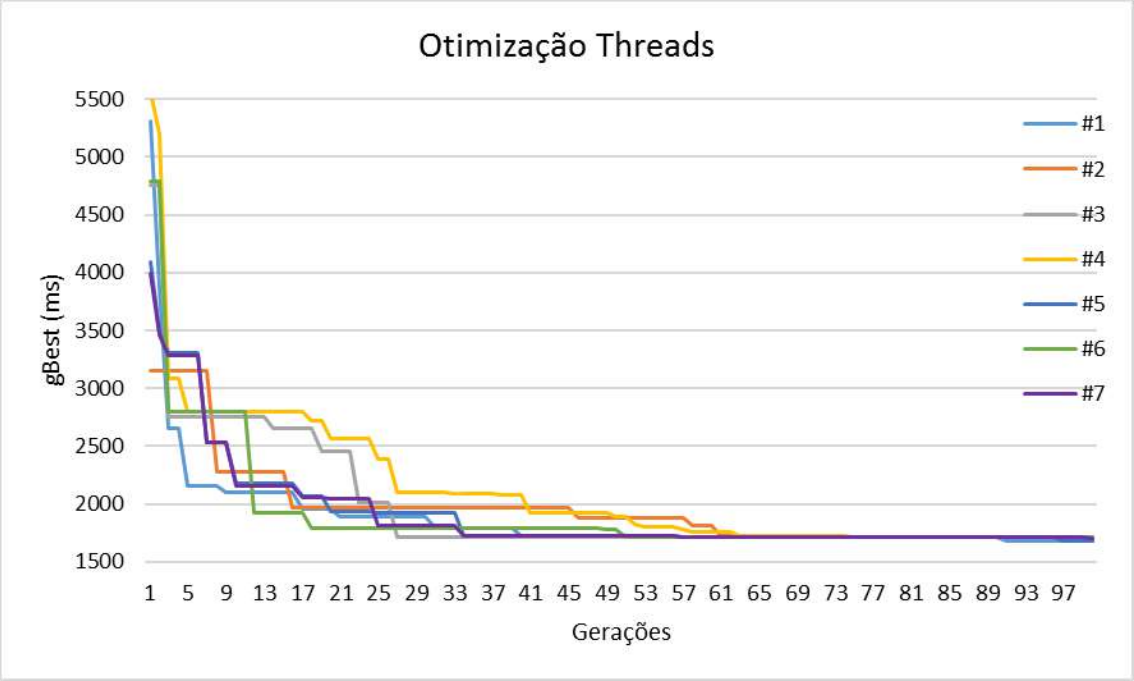

Figura 114 – Gerações versus gBest (Otimização - zoom)

A Figura 115 exibe o histograma típico das 10 combinações de threads por blocos (X, Y, Z) mais encontradas nos testes. Aqui, novamente pode-se observar que para a grande maioria dos experimentos, tem-se quase 50% de diferença entre a quantidade de combinações encontradas do primeiro lugar para o segundo.

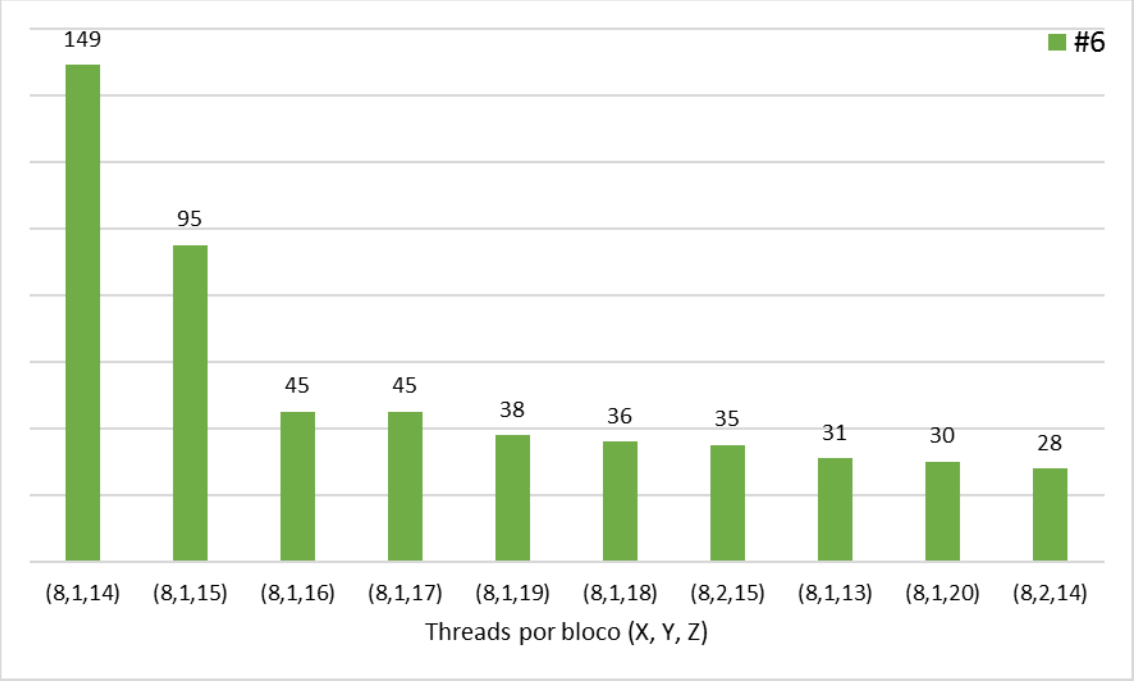

Figura 115 – Histograma das 10 combinações mais encontradas (Otimização)

Após a geração do histograma individual de cada experiência, foi gerado um histograma (Figura 116) contendo os dois melhores gBests de cada experimento, independente de quantas vezes esta combinação de threads por bloco se repetia no experimento.

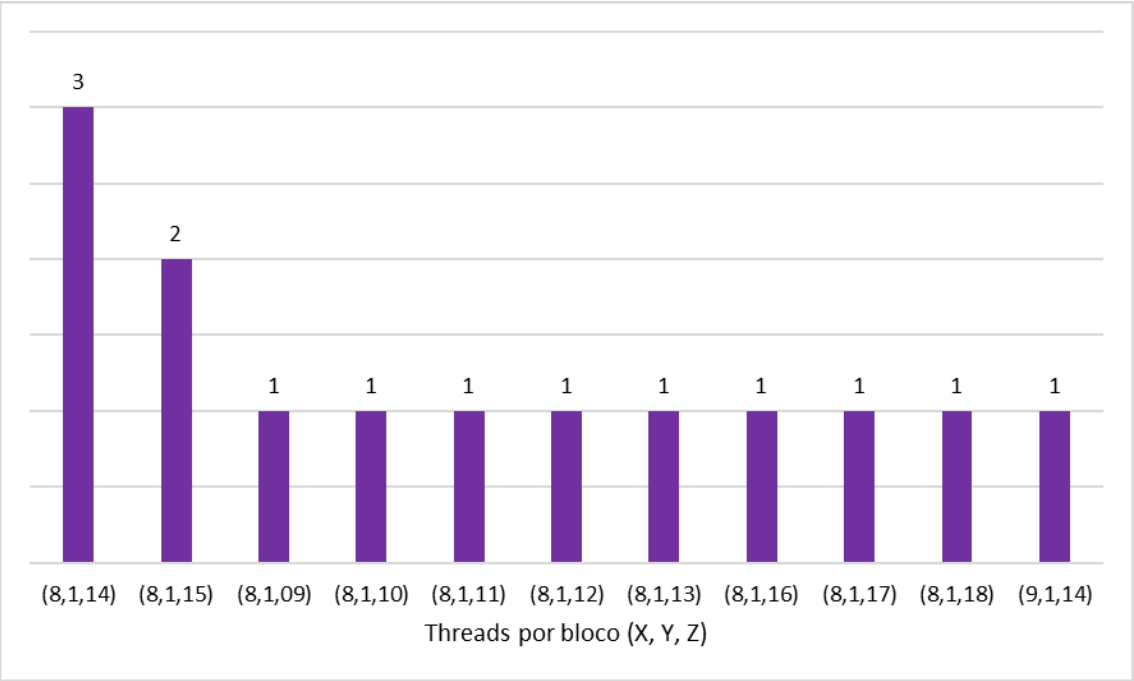

Figura 116 – Histograma das combinações mais encontradas em cada experiência

Assim, percebe-se que a combinação de threads por bloco (8, 1, 14) se repete três vezes nas quatorze melhores combinações encontradas, o que corresponde a 22% destas combinações, sendo seguido pela combinação (8, 1, 15) com 15%. A Figura 117 ilustra as porcentagens de repetição das melhores combinações encontradas pelo algoritmo PSO nos sete experimentos executados.

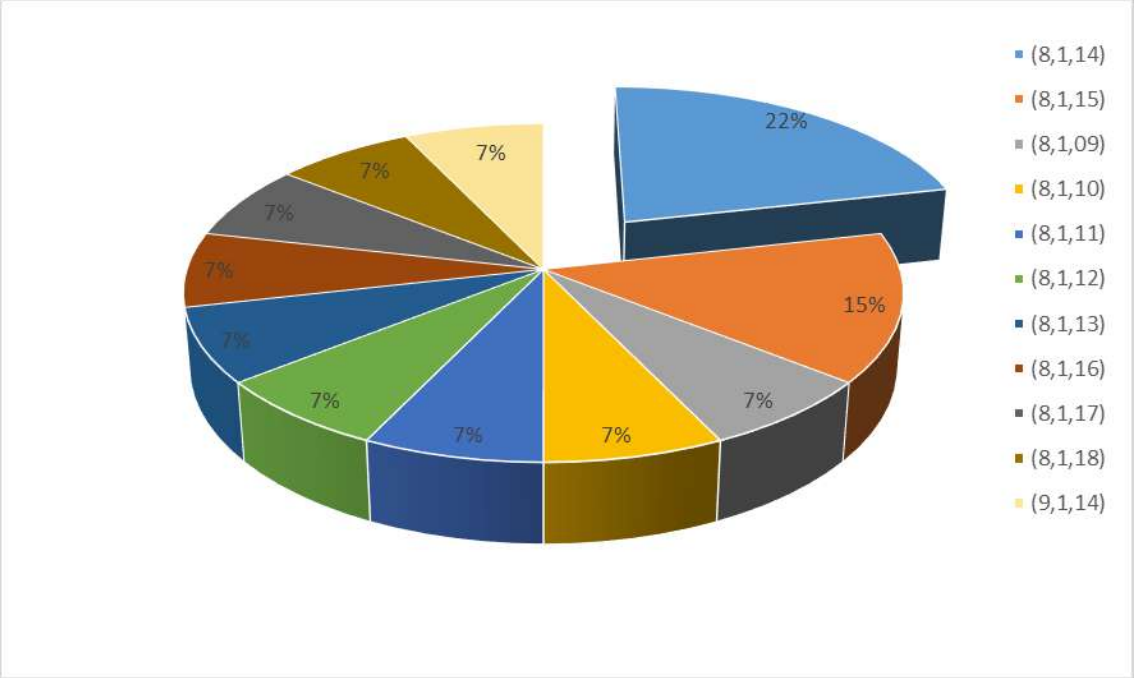

Figura 117 – Porcentagem de repetição das melhores combinações encontradas

Levando em consideração o espaço de busca possuindo 262.144 possibilidades (64 x 64 x 64), verificou-se através dos experimentos aqui realizados que, pouquíssimas combinações conseguem minimizar de forma adequada o tempo da rotina WEST do Campo de Vento. Neste trabalho foi escolhida a combinação de threads por bloco (8, 1, 14) para todos os demais experimentos.

4.2.4. Consistência da abordagem

Antes da quantificação dos *speedups* e ganhos devido à otimização da alocação dos threads via PSO, o algoritmo proposto foi aplicado em todos os 20 ventos reais observados. Diversos experimentos comparativos entre os resultados obtidos deste algoritmo e os resultados obtidos com o algoritmo anterior foram feitos e, como resultado de tais experimentos, foram produzidos resultados idênticos ao algoritmo anterior.

4.2.5. Resultados

Considerando que o algoritmo paralelo com a realocação dos processadores ociosos e com a otimização da alocação dos threads via PSO já está consistente, o objetivo desta seção é quantificar e analisar os tempos de execução dos *speedups* obtidos com o uso do programa baseado em GPU com a realocação dos processadores ociosos e com a otimização da alocação dos threads via PSO. Para isso, foi escolhido um único vento observado (Vento #2) e simulações com todos os níveis de refinamento (R1, R4, R16 e R64) e vários números de iterações (500, 1000, 1500 e 2000) foram investigados.

A Tabela 25 mostra os resultados comparativos entre os tempos de execução (em segundos) da rotina WEST do módulo de Campo de Vento (média de 6 execuções) para as implementações sequenciais (CPU), primeira implementação paralela (GPU_1), para a implementação paralela com a realocação dos processadores ociosos (GPU_2) e para a realocação dos processadores ociosos e com a otimização da alocação dos threads via PSO (GPU_3) para os diferentes níveis de refinamento e número de iterações.

Pode-se observar um aumento considerável do *speedup* desta implementação ($Speedup_3$) em relação a implementação anterior ($Speedup_2$), demonstrando assim que a otimização da alocação dos threads via PSO obteve êxito no desempenho do algoritmo. As mesmas características de hardware utilizados na primeira implementação paralela, aqui foram também utilizadas. Os tempos calculados para as versões de GPU novamente incluem a execução de kernels na GPU, alocação de memória na GPU e transferência de dados para a GPU.

500							
	CPU	**GPU$_1$***	**GPU$_2$***	**GPU$_3$***	**Speedup$_1$**	**Speedup$_2$**	**Speedup$_3$**
R1	0,52	0,25	0,24	0,19	2,04	2,22	2,80
R4	3,00	0,46	0,40	0,26	6,55	7,44	11,46
R16	24,91	1,30	1,12	0,54	19,07	22,28	46,57
R64	112,24	4,66	4,13	1,62	24,09	27,21	69,36
1000							
	CPU	**GPU$_1$***	**GPU$_2$***	**GPU$_3$***	**Speedup$_1$**	**Speedup$_2$**	**Speedup$_3$**
R1	1,02	0,40	0,35	0,28	2,56	2,94	3,61
R4	5,94	0,81	0,66	0,41	7,34	8,95	14,48
R16	49,74	2,47	2,08	0,92	20,13	23,89	53,90
R64	223,68	9,09	7,98	3,01	24,62	28,02	74,23
1500							
	CPU	**GPU$_1$***	**GPU$_2$***	**GPU$_3$***	**Speedup$_1$**	**Speedup$_2$**	**Speedup$_3$**
R1	1,52	0,51	0,49	0,37	2,96	3,10	4,07
R4	10,02	1,16	0,94	0,56	8,63	10,70	17,95
R16	74,67	3,65	3,05	1,34	20,45	24,48	55,93
R64	334,67	13,52	11,86	4,37	24,76	28,21	76,65
2000							
	CPU	**GPU$_1$***	**GPU$_2$***	**GPU$_3$***	**Speedup$_1$**	**Speedup$_2$**	**Speedup$_3$**
R1	2,02	0,66	0,64	0,44	3,05	3,18	4,58
R4	13,52	1,51	1,26	0,72	8,95	10,75	18,87
R16	98,76	4,82	4,01	1,71	20,49	24,66	57,81
R64	449,29	18,04	15,15	5,60	24,91	29,66	80,27

*Considerando kernel GPU + alocação de memória na GPU + transferência de dados para a GPU

Tabela 25 - *Speedups* e tempos de execução das implementações sequenciais e paralelas

A Figura 118 até a Figura 121, mostram a influência do número de iterações no tempo de execução para todos os domínios computacionais e para as diferentes implementações do algoritmo. Observa-se que mesmo utilizando a combinação de threads por bloco otimizada para o nível de resolução R16, tem-se um ganho significativo nos demais níveis de refinamento.

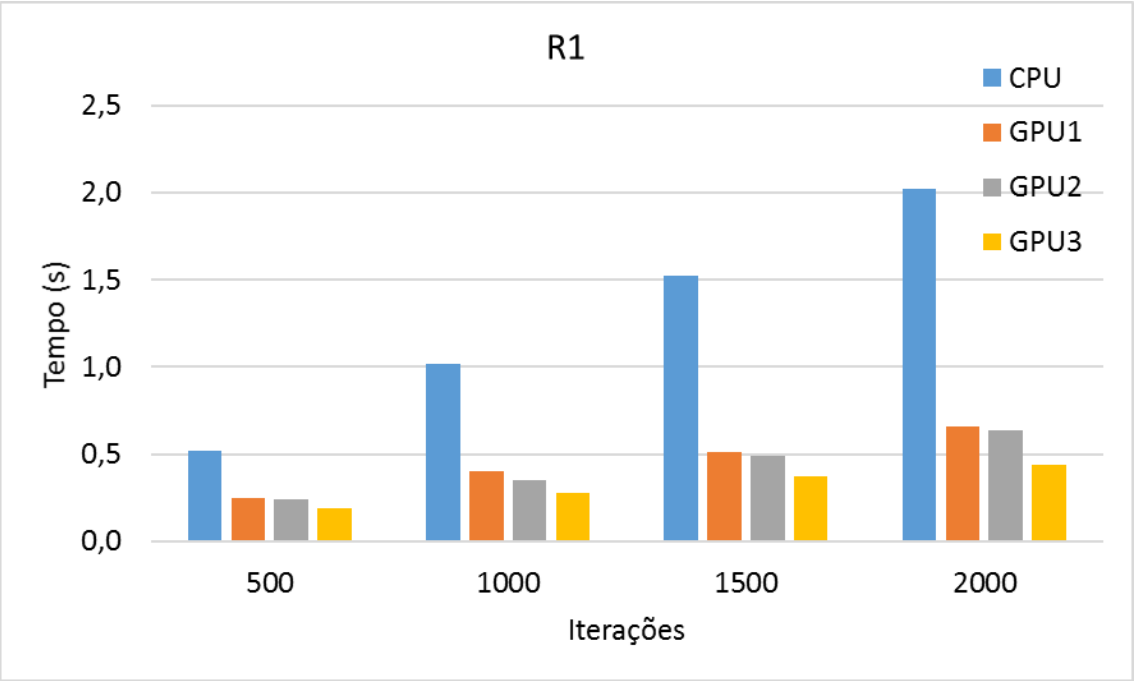

Figura 118 – Tempo de execução versus número de iterações (R1)

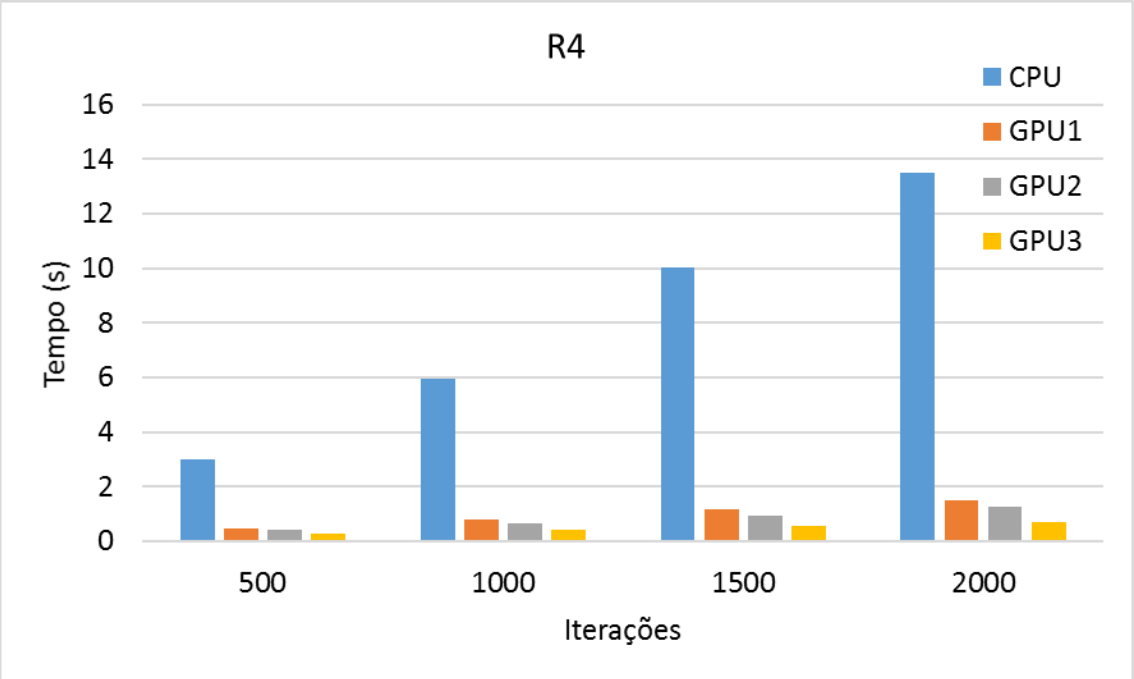

Figura 119 – Tempo de execução versus número de iterações (R4)

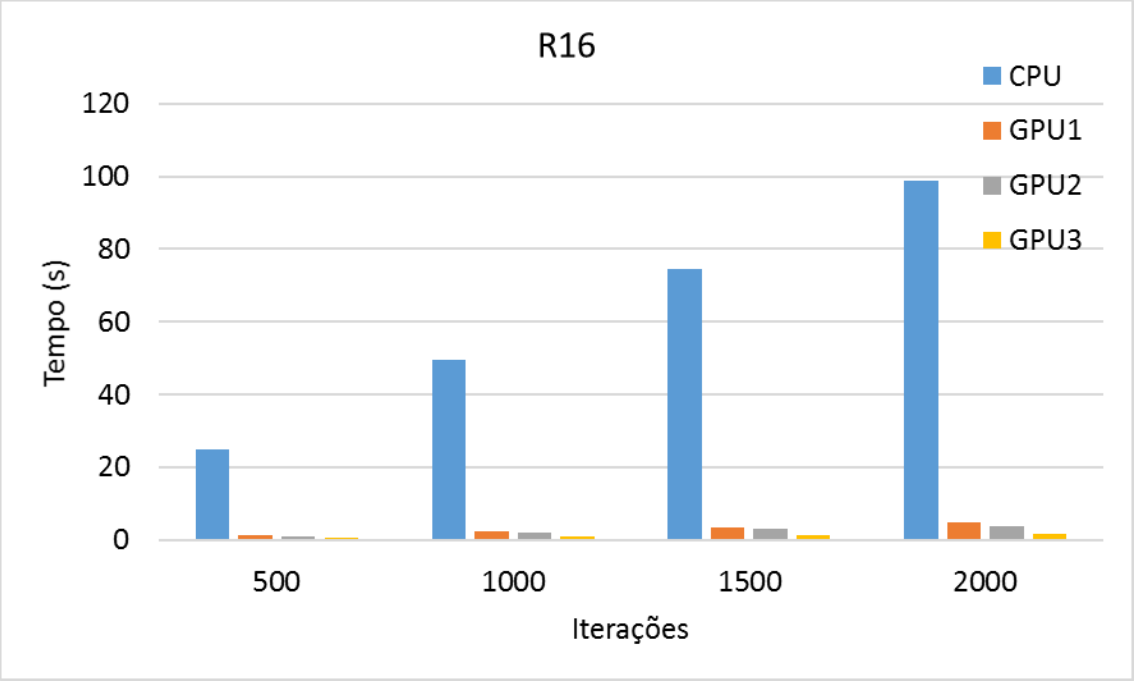

Figura 120 – Tempo de execução versus número de iterações (R16)

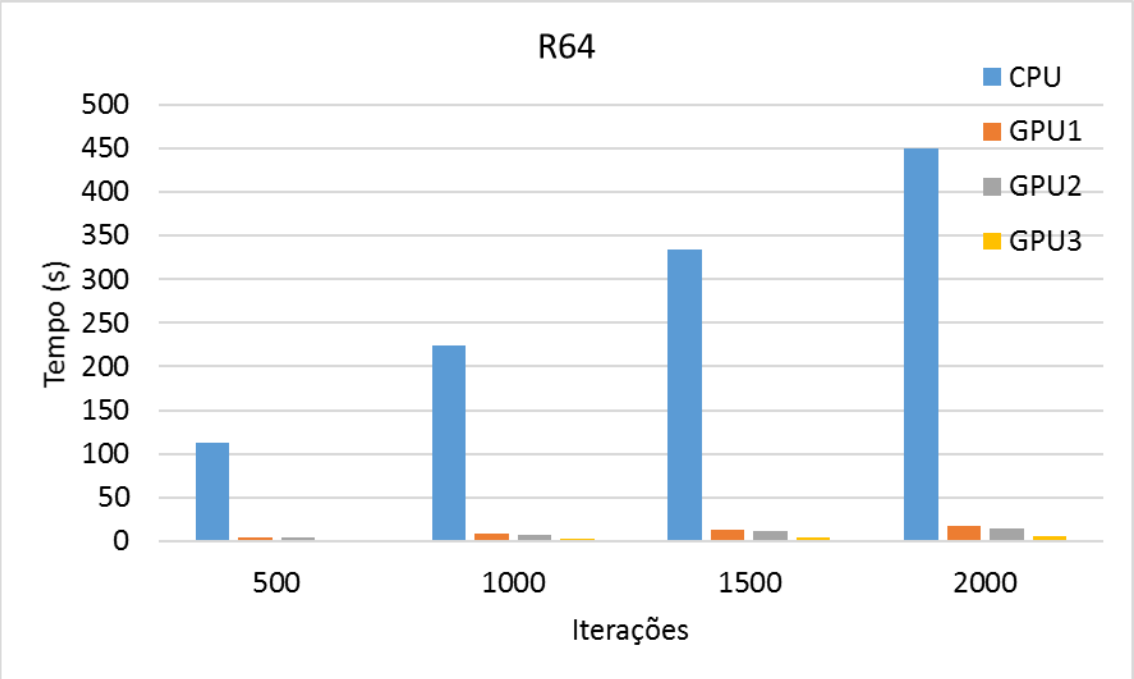

Figura 121 – Tempo de execução versus número de iterações (R64)

Devido aos grandes valores atingidos pela rotina WEST com o uso da CPU, apresenta-se novamente (na Figura 122 até a Figura 125) a influência do tempo de

execução com o aumento do numero de iterações entretanto, foi retirado o tempo da CPU para uma melhor visualização do tempo ganho com o número de threads por blocos otimizado.

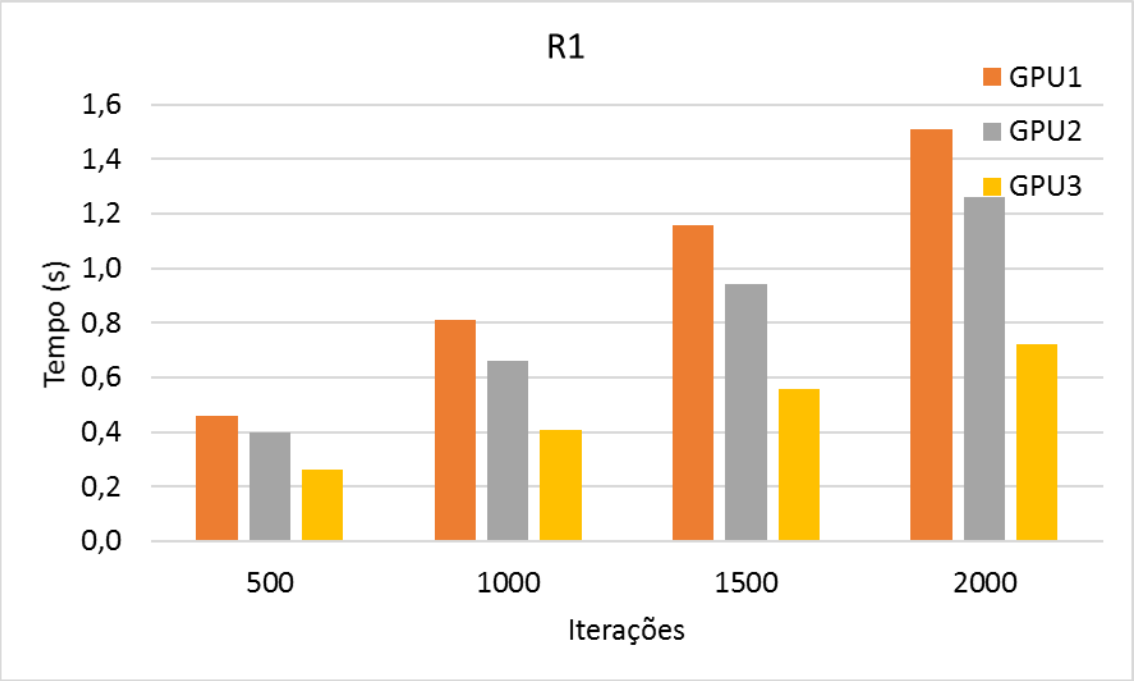

Figura 122 – Tempo de execução versus número de iterações (sem CPU)

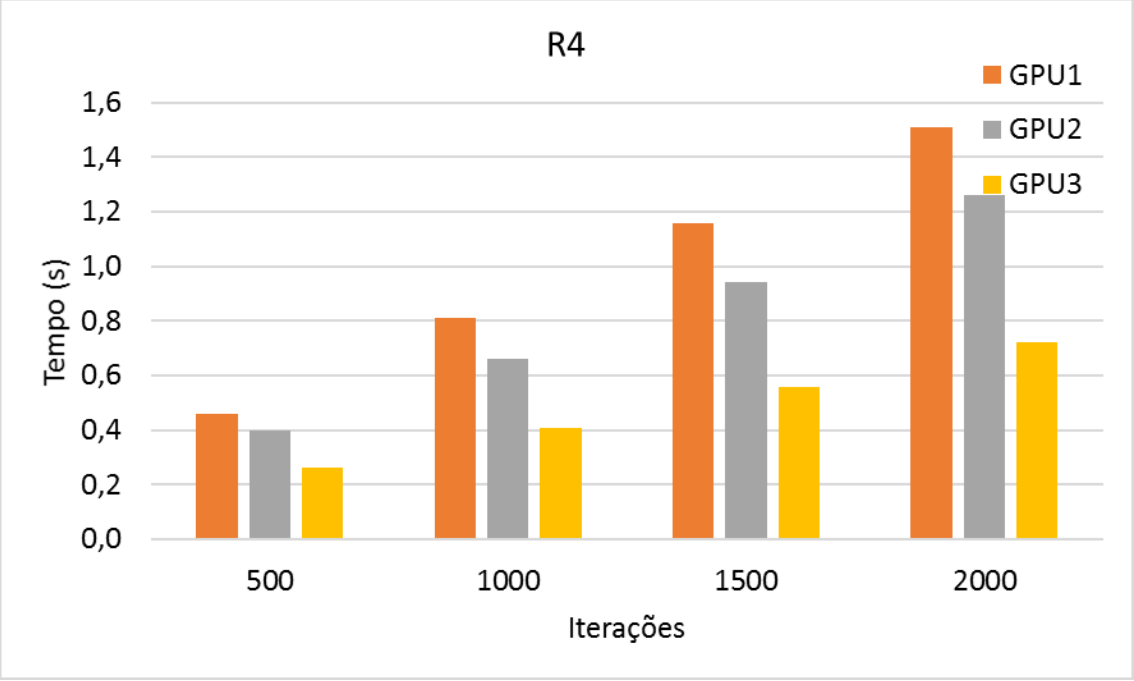

Figura 123 – Tempo de execução versus número de iterações (sem CPU – R4)

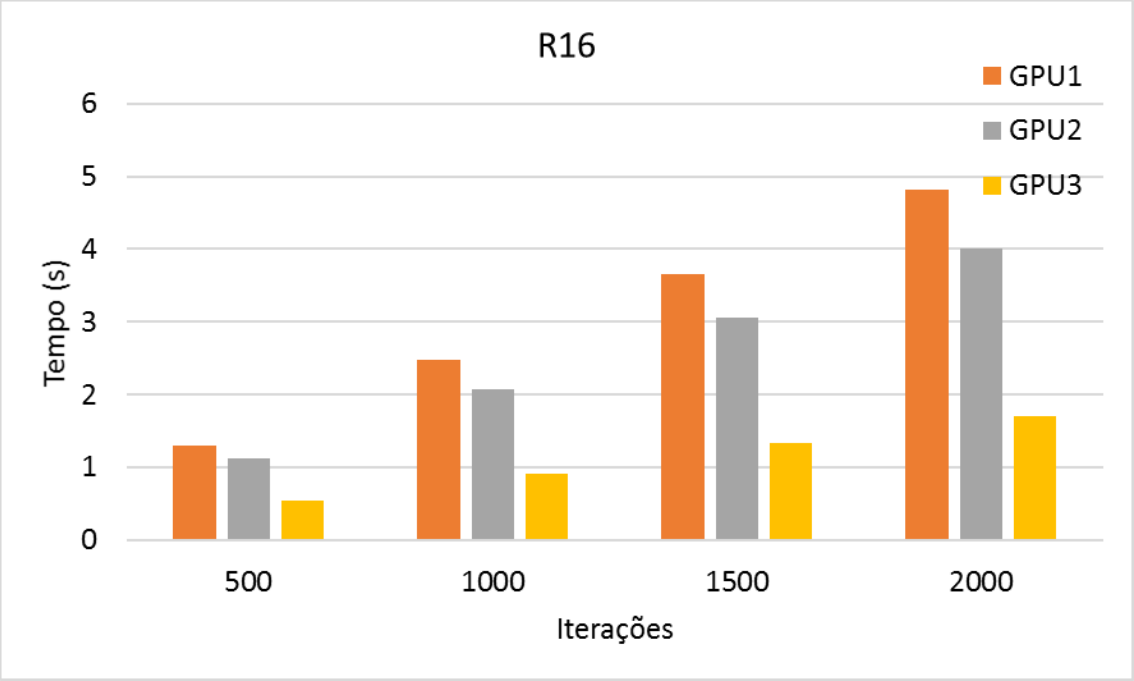

Figura 124 – Tempo de execução versus número de iterações (sem CPU – R16)

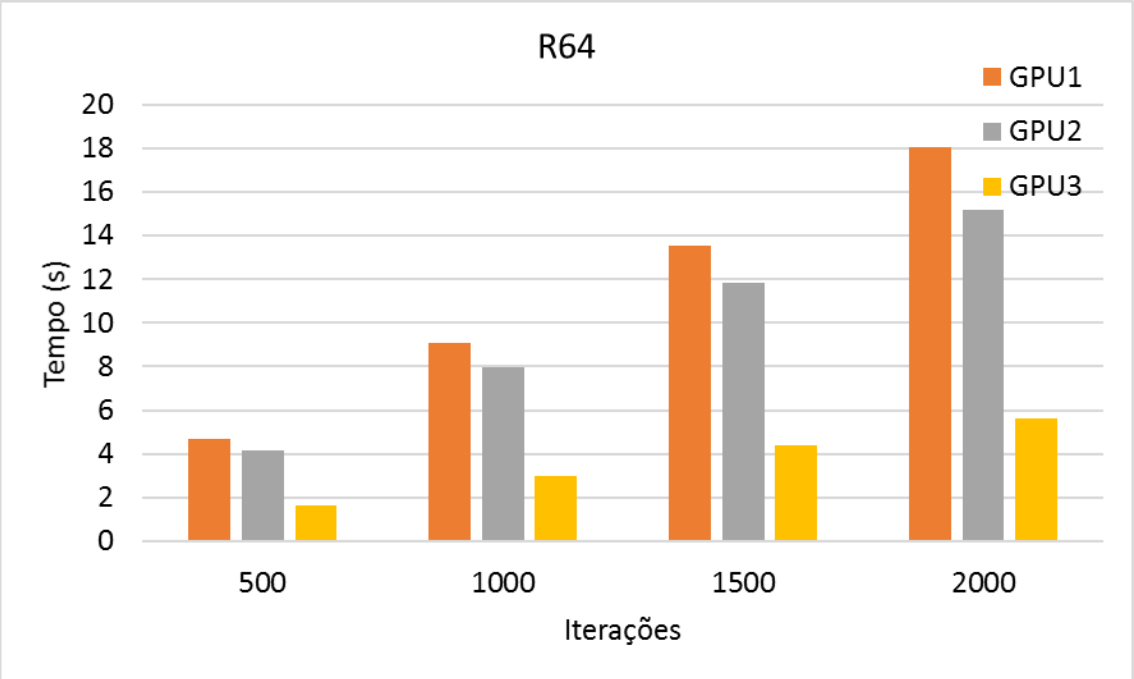

Figura 125 – Tempo de execução versus número de iterações (sem CPU – R64)

Com o aumento das iterações e do domínio computacional, tem-se um aumento do *speedup*, entretanto, observando da Figura 126 à Figura 129, fica evidente como este acréscimo passa a ser significativo após a otimização dos threads por blocos pelo PSO.

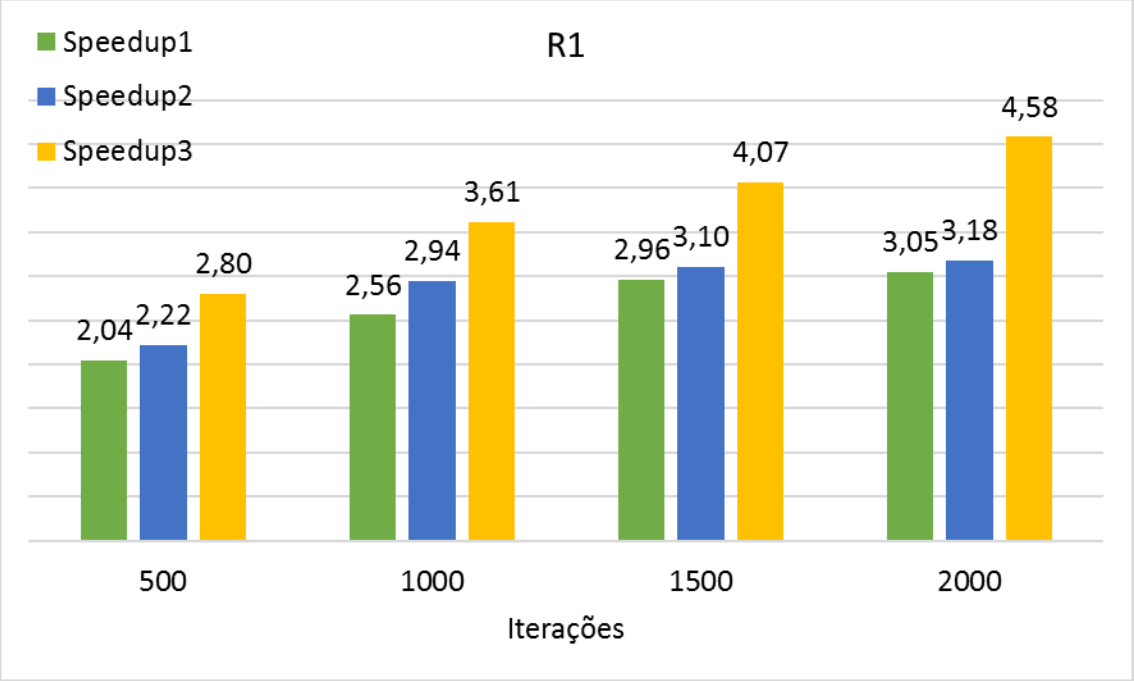

Figura 126 – *Speedup* versus número de iterações (R1)

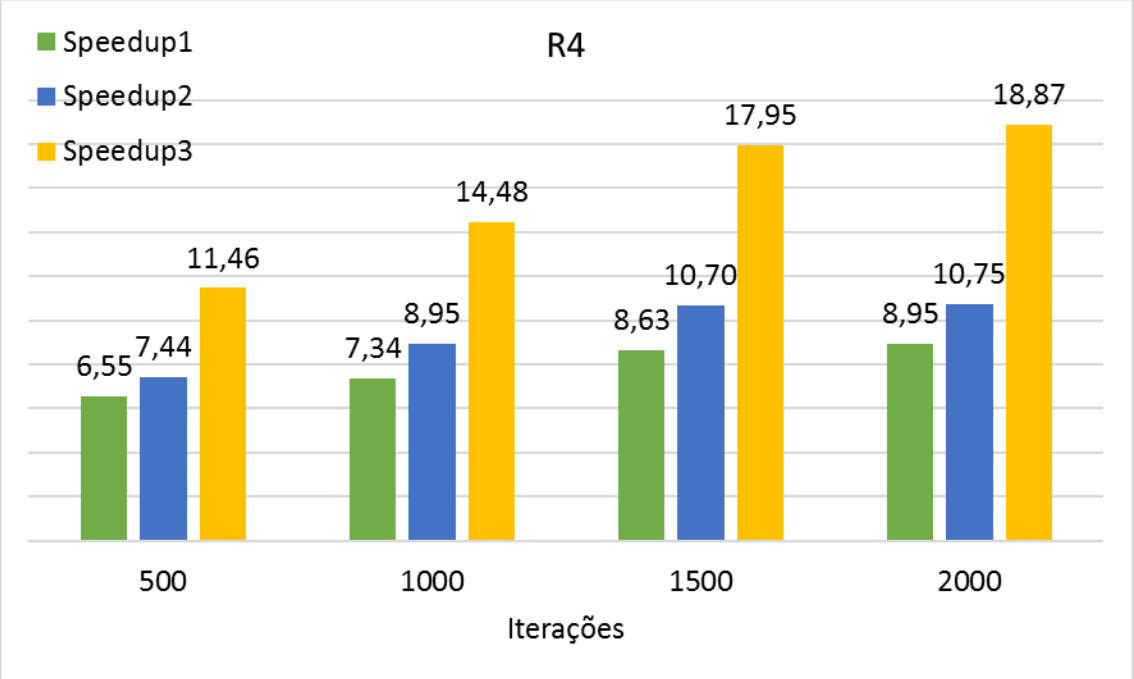

Figura 127 – *Speedup* versus número de iterações (R4)

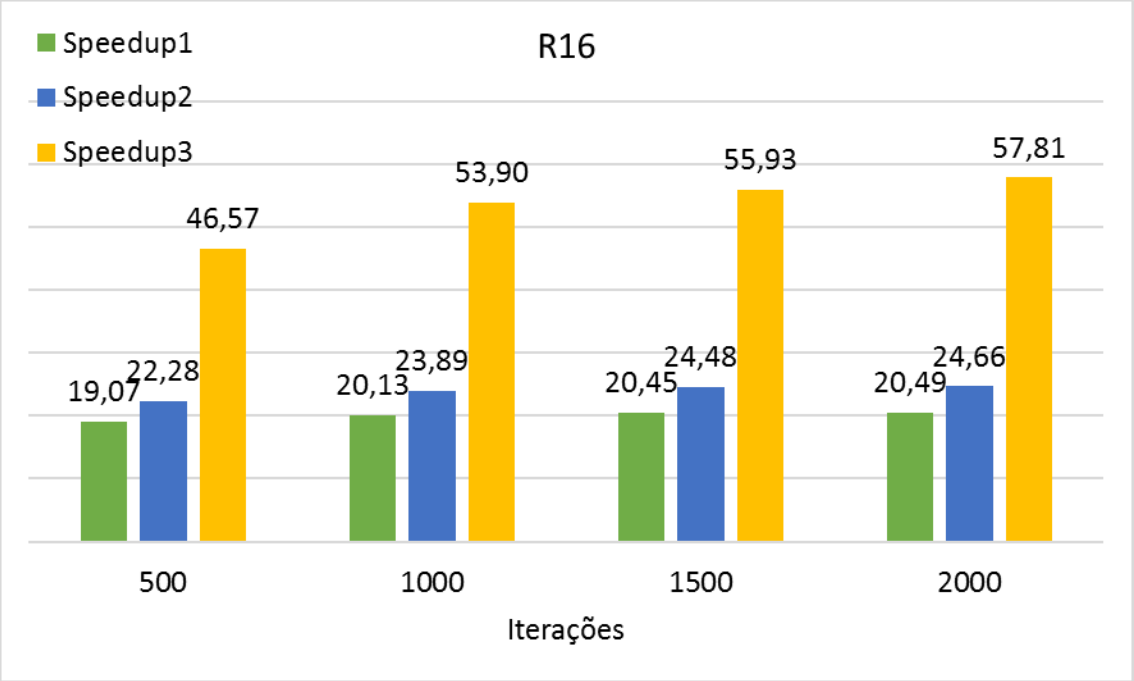

Figura 128 – *Speedup* versus número de iterações (R16)

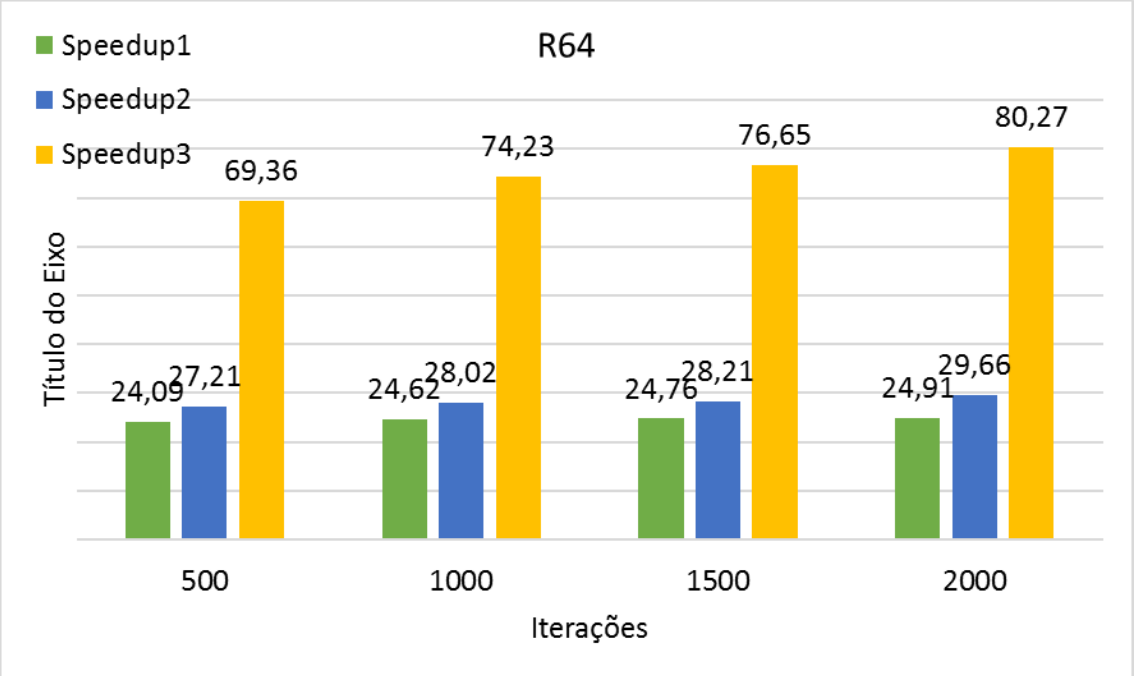

Figura 129 – *Speedup* versus número de iterações (R64)

Da Figura 130 à Figura 133, pode-se observar o acréscimo do tempo de execução com o aumento do nível de refinamento.

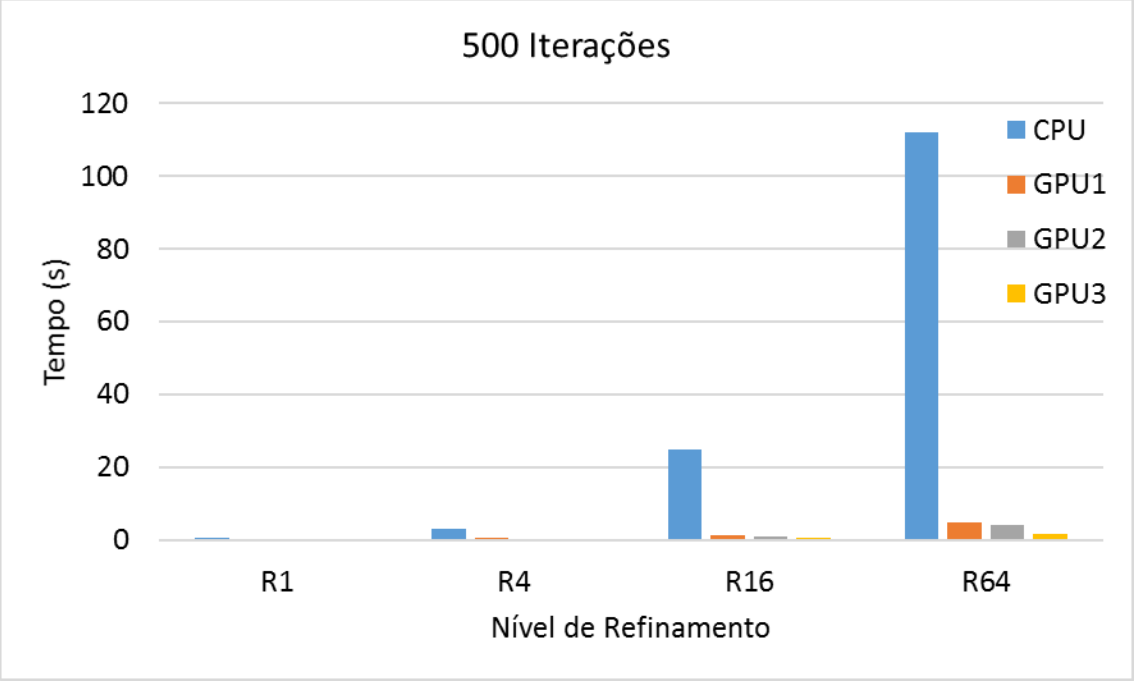

Figura 130 – Tempo de execução versus nível de refinamento (500 Iterações)

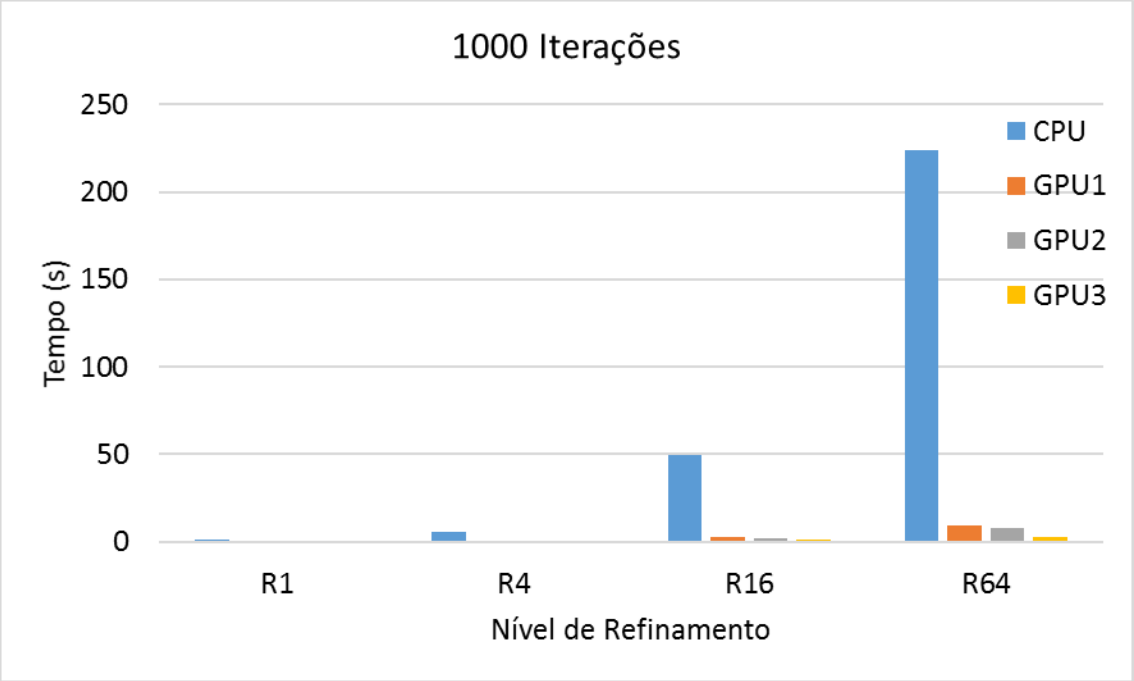

Figura 131 – Tempo de execução versus nível de refinamento (1000 Iterações)

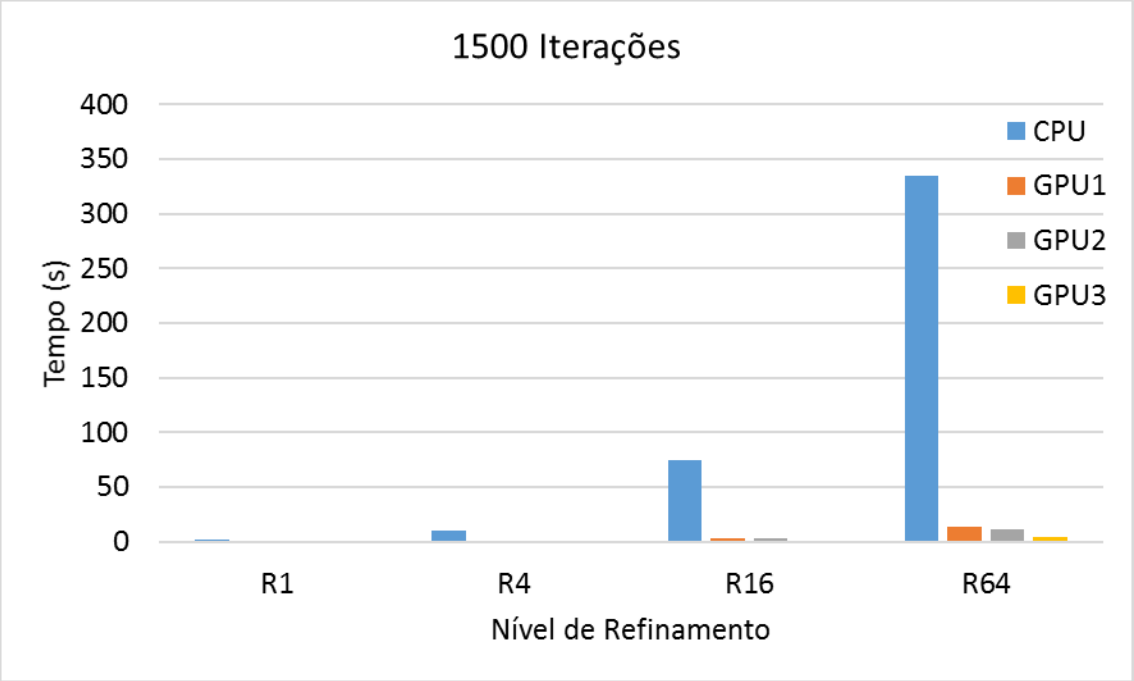

Figura 132 – Tempo de execução versus nível de refinamento (1500 Iterações)

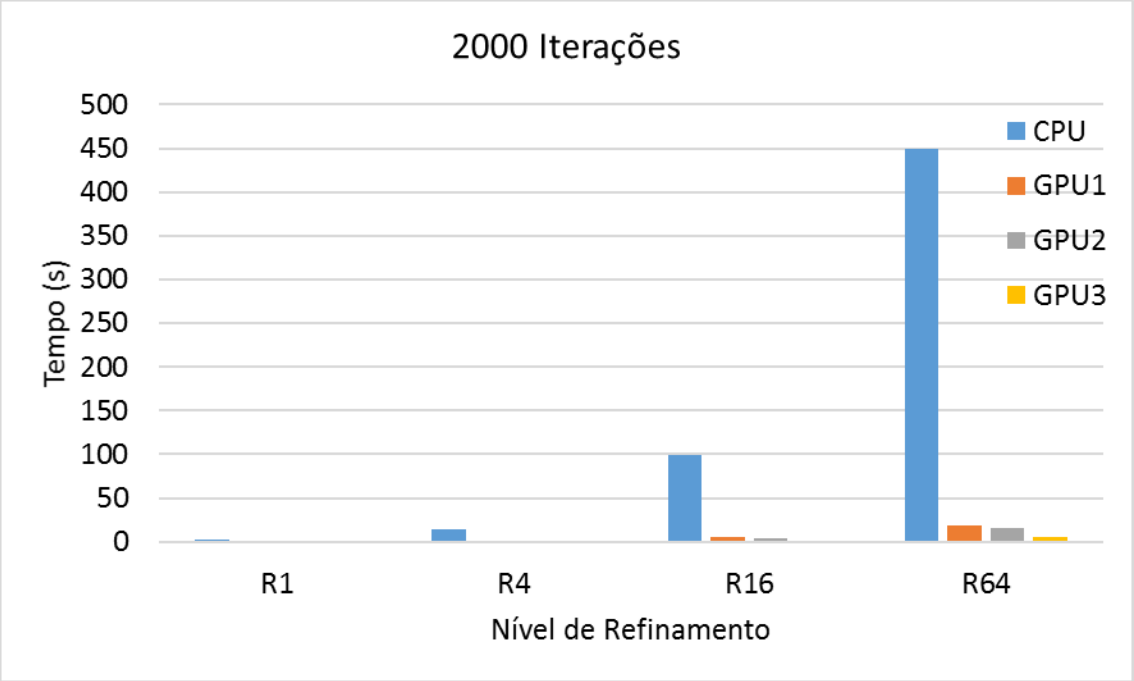

Figura 133 – Tempo de execução versus nível de refinamento (2000 Iterações)

Para poder investigar melhor o ganho com a otimização dos threads por blocos, o tempo de CPU foi retirado nos mostrando assim da Figura 134 até a Figura 137, o acréscimo do tempo de processamento sem o tempo de CPU.

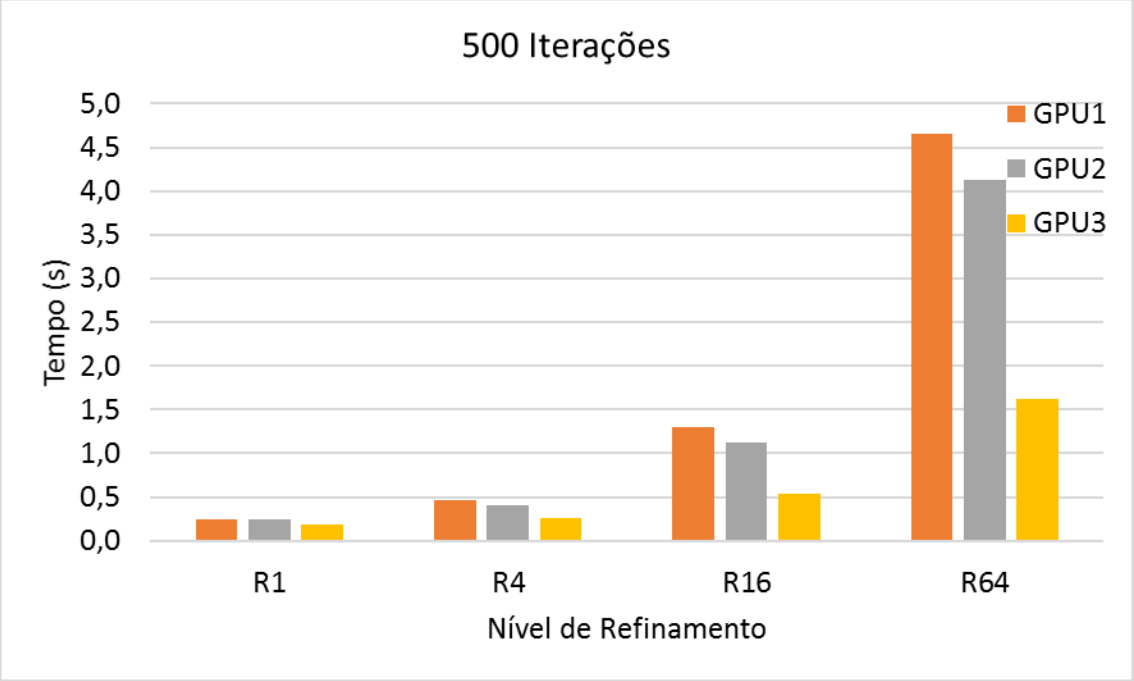

Figura 134 – Tempo de execução versus nível de refinamento (sem CPU – 500 Iterações)

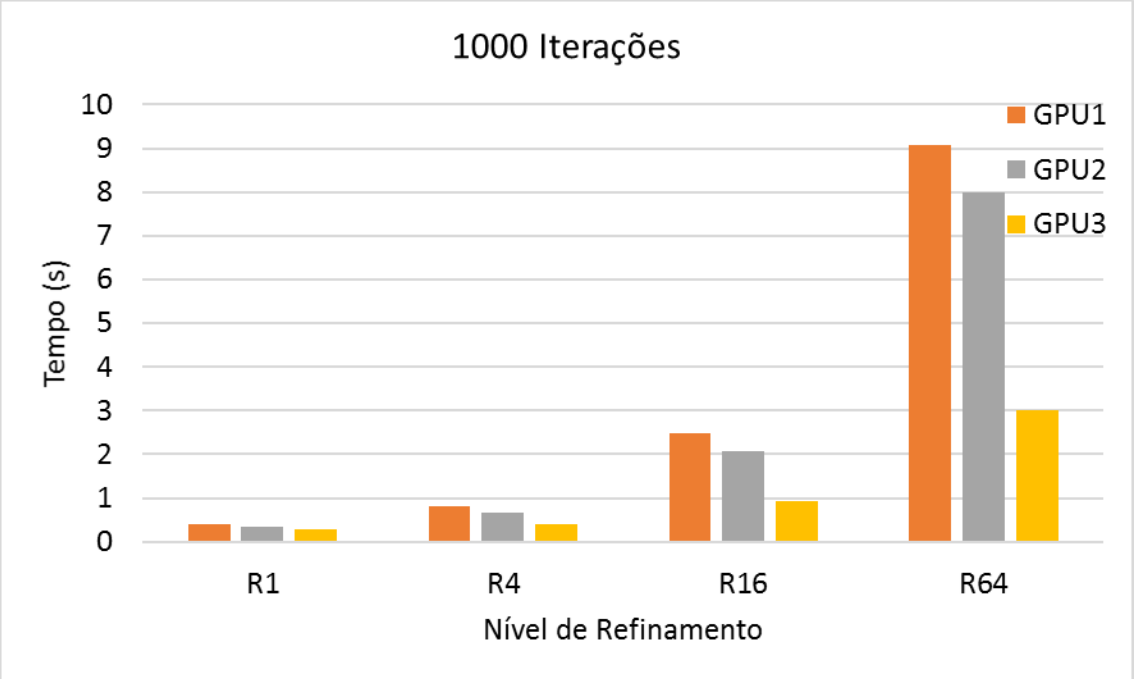

Figura 135 – Tempo de execução versus nível de refinamento (sem CPU – 1000 Iterações)

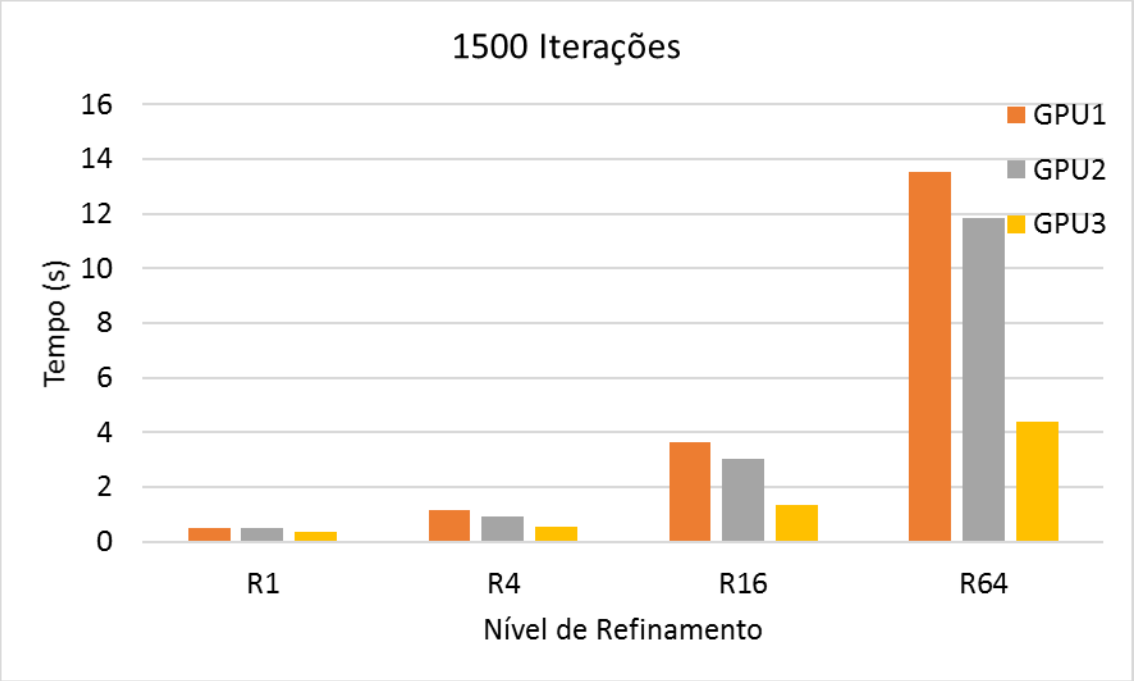

Figura 136 – Tempo de execução versus nível de refinamento (sem CPU – 1500 Iterações)

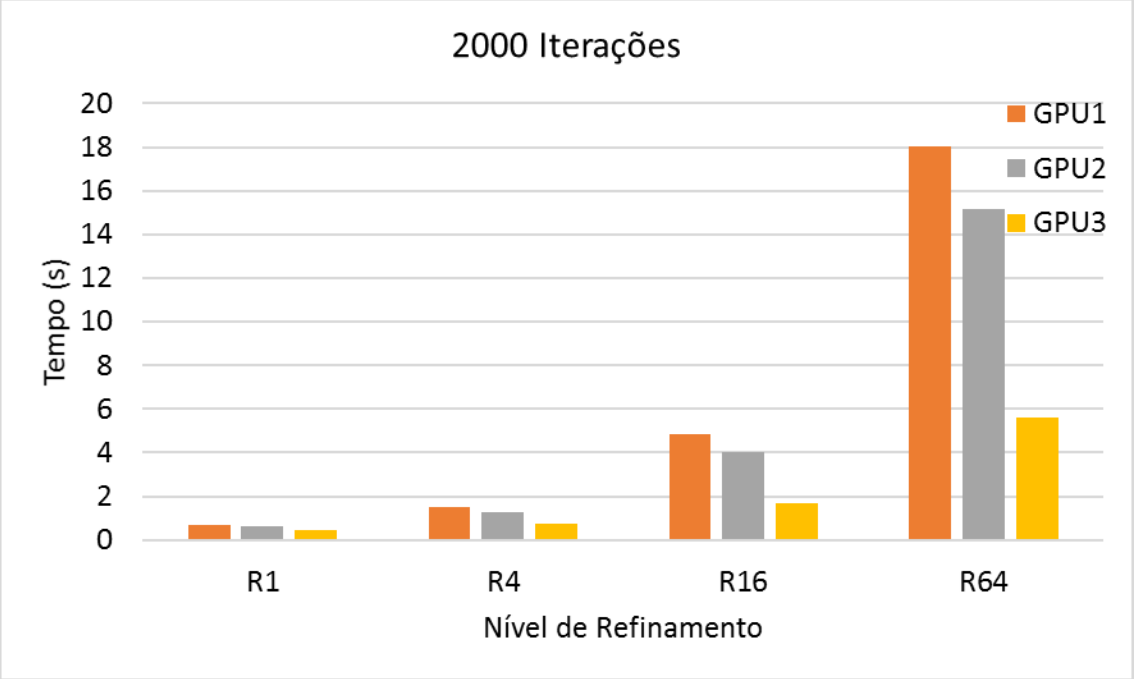

Figura 137 – Tempo de execução versus nível de refinamento (sem CPU – 2000 Iterações)

A Figura 138 até a Figura 141, mostram os gráficos comparativos das acelerações com o refinamento do domínio computacional, para todas as diferentes iterações. Pode-se observar o grande acréscimo do *speedup* depois da quantidade de threads por blocos otimizado.

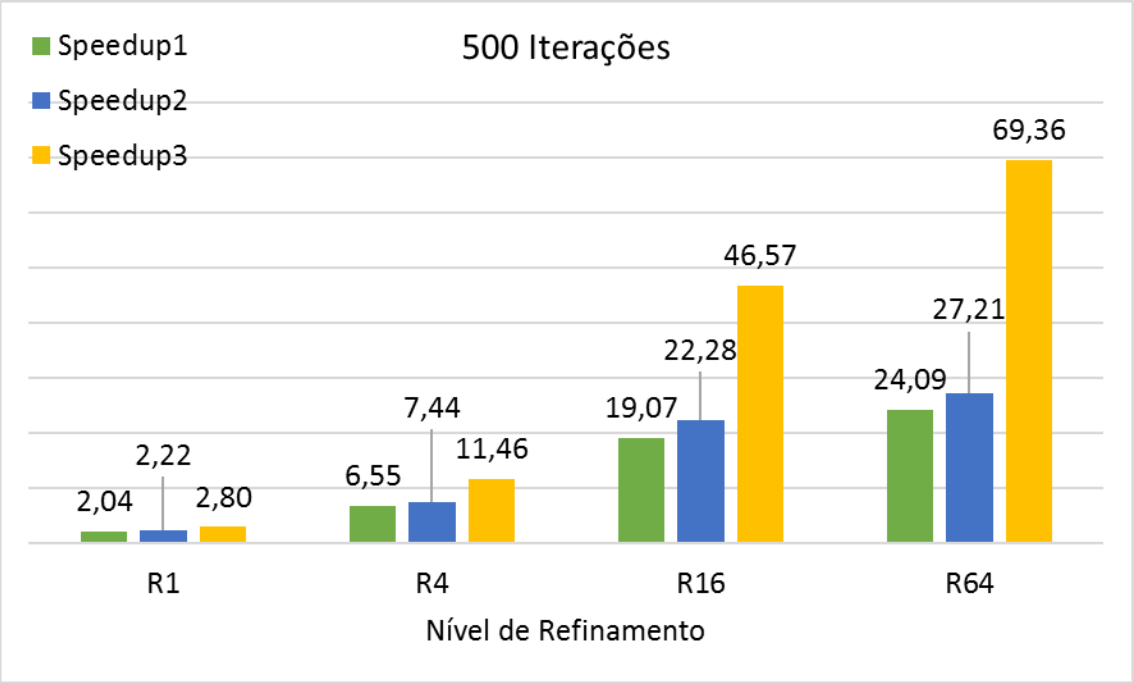

Figura 138 – *Speedups* para diferentes níveis de refinamentos (500 Iterações)

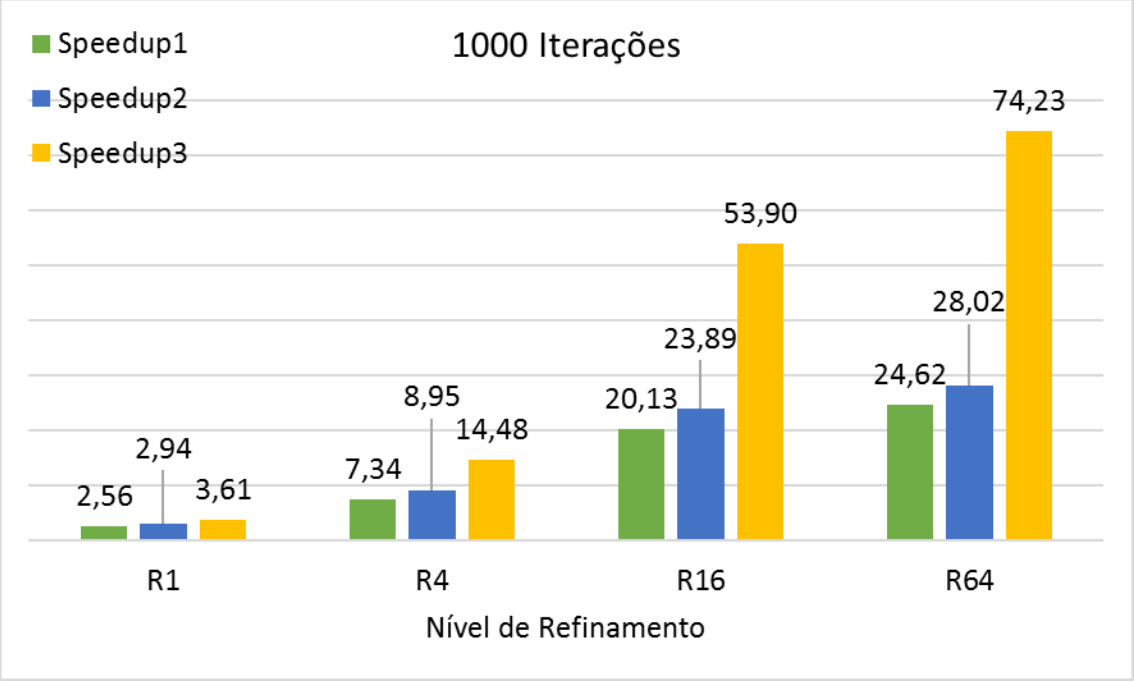

Figura 139 – *Speedups* para diferentes níveis de refinamentos (1000 Iterações)

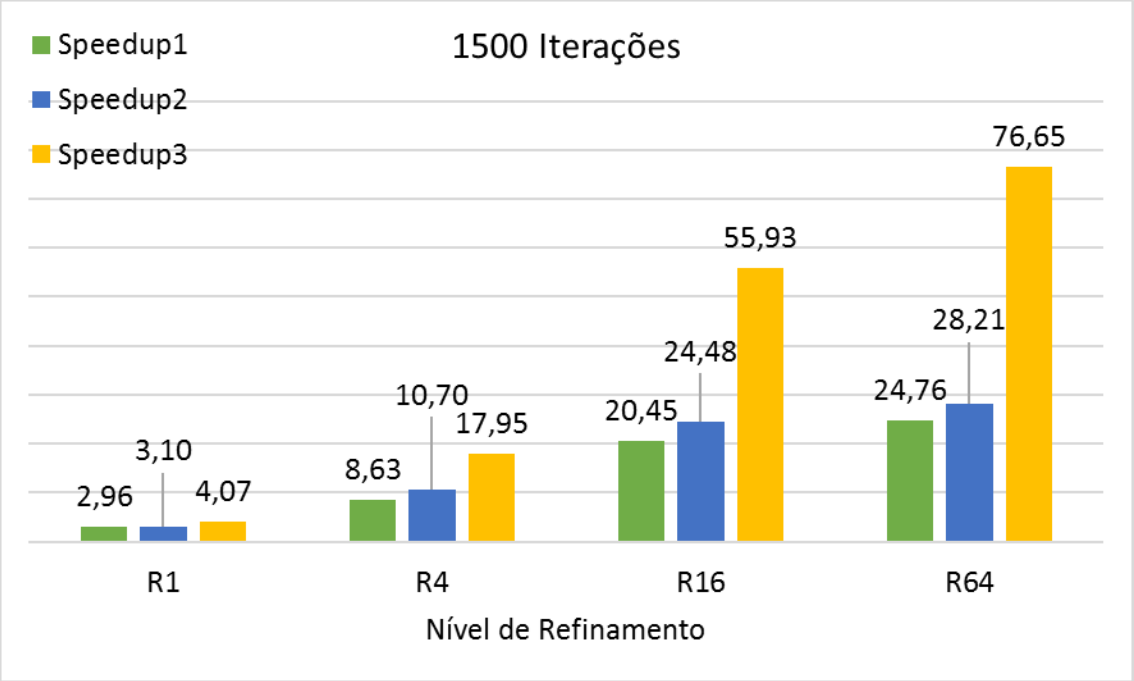

Figura 140 – *Speedups* para diferentes níveis de refinamentos (1500 Iterações)

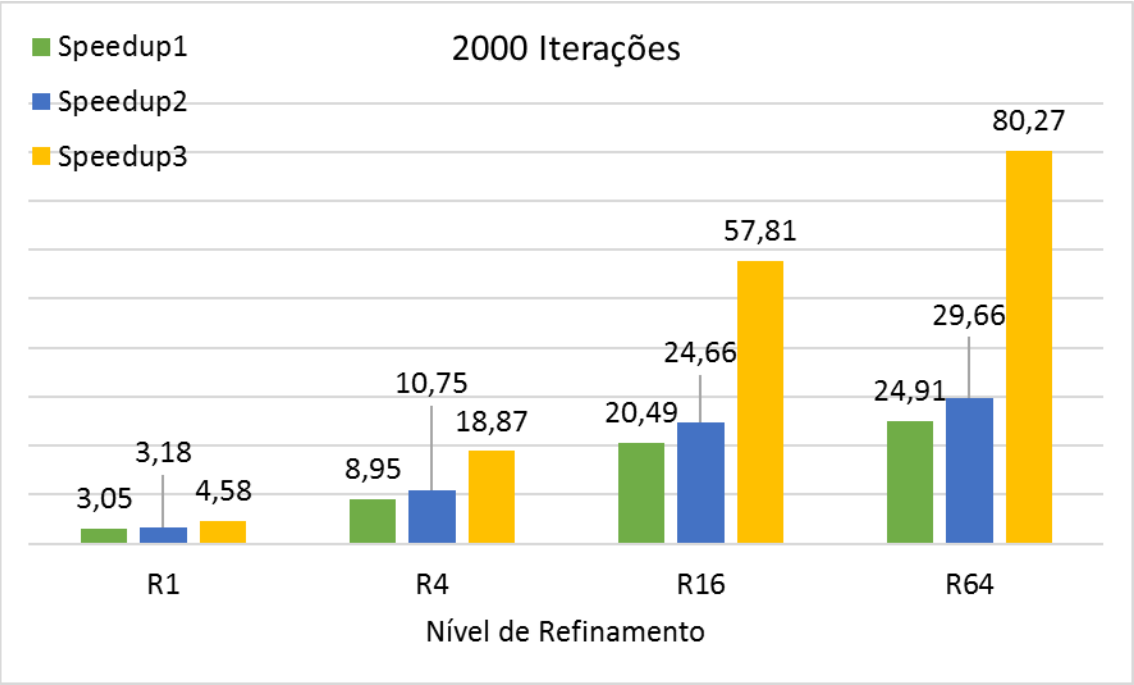

Figura 141 – *Speedups* para diferentes níveis de refinamentos (2000 Iterações)

Após os diversos testes realizados, pode-se notar que embora a otimização com o uso de GPU do algoritmo do Campo de Vento tenha melhorado seu desempenho, faz-se

necessário o correto dimensionamento de threads por blocos, pois proporciona um desempenho muito maior na execução do algoritmo.

4.2.6. Outros níveis de refinamento

Devido aos motivadores valores de *speedup* alcançados após a otimização dos threads pelo PSO nos experimentos anteriores, foram implementados mais dois níveis de refinamento, R256 e R1024. Nestes novos níveis foram feitos experimentos comparativos somente para o vento #2 entre o algoritmo executados na CPU e o algoritmo executado na GPU com otimização dos threads por PSO. Como resultado de tais experimentos, foram produzidos resultados semelhantes (diferenças de 10^{-10}) devido a diferença de arredondamento de CPU e GPU como já mencionado.

Para os novos níveis de refinamento foram utilizadas as mesmas configurações de threads por blocos encontradas pelo algoritmo PSO para o nível de refinamento R16 (8, 1, 14). Novamente pode-se comprovar a performance do algoritmo e acredita-se que se o algoritmo PSO for executado com estes níveis de refinamento, pode-se obter valores maiores de *speedup*. Tais testes não foram executados pois, como dito anteriormente, consomem uma enorme quantidade de tempo e, o nível adequado de refinamento ainda não é conhecido. Tais testes foram feitos para que se possa ter conhecimento da ordem de grandeza do tempo necessário caso um destes níveis de refinamento seja o mais indicado para a execução do SCA.

A Tabela 26 mostra os resultados comparativos entre os tempos de execução (em segundos) da rotina WEST do módulo de Campo de Vento (apenas 1 execução) para a implementação sequencial (CPU) e a implementação com realocação dos processadores ociosos e com a otimização da alocação dos threads via PSO (GPU_3) para os novos níveis

de refinamento e número de iterações. Novamente, os tempos calculados para as versões de GPU incluem a execução de kernels na GPU, alocação de memória na GPU e transferência de dados para a GPU.

500			
	CPU	**GPU$_3$***	**Speedup$_3$**
R256	561,04	6,09	92,12
R1024	2322,72	23,93	97,07
1000			
	CPU	**GPU$_3$***	**Speedup$_3$**
R256	1125,43	11,51	97,75
R1024	4618,89	45,79	100,88
1500			
	CPU	**GPU$_3$***	**Speedup$_3$**
R256	1667,89	17,03	97,93
R1024	6911,09	67,63	102,18
2000			
	CPU	**GPU$_3$***	**Speedup$_3$**
R256	2224,92	22,51	98,86
R1024	9203,63	89,43	102,91

*Considerando kernel GPU + alocação de memória na GPU + transferência de dados para a GPU

Tabela 26 – *Speedups* e tempos de execução das implementações sequenciais e paralelas dos novos níveis de refinamento

4.3. AVALIAÇÃO QUALITATIVA CONSIDERANDO OS TRABALHOS RELACIONADOS

Comparando os resultados obtidos neste trabalho com os apresentados na literatura atual seria muito difícil e talvez impreciso, devido a diferenças intrínsecas entre modelos, métodos numéricos, implementações e plataformas de computadores. No entanto, optou-se por avaliar qualitativamente os ganhos obtidos aqui à luz dos resultados da literatura.

Como pode ser visto anteriormente, poucos resultados numéricos de *speedups* foram encontrados para o cálculo do Campo de Vento usando computação paralela (

(SANJUAN, MARGALEF e CORTÉS, 2016) e (MEDINA e CORTÉS, 2015)). A Tabela 27 resume as velocidades numéricas fornecidas por diferentes autores, mostrando o número de células no domínio computacional (Células), o tempo de execução, em segundos, das abordagens sequencial (T_{SEQ}) e paralela (T_{PAR}), o *speedup* (Spup) e algumas características de *hardware*.

Referência	**Células**	**T_{SEQ}**	**T_{PAR}**	**Spup**	**Hardware**
Sanjuan, 2016	640.000	496,78	90,20	5,5	PC x 10 nós clusters
Medina and Cortez, 2015	800.000	204,32	5,17	39,5	PC x GTX Titan
Medina and Cortez, 2015	1.800.000	371,59	8,48	43,8	PC x GTX Titan
Trabalho atual (1)	1.475.072	449,29	18,04	24,90	PC-Intel-I7 x GTX-680
Trabalho atual (2)	1.475.072	449,29	15,15	29,66	PC-Intel-I7 x GTX-680
Trabalho atual (3)	1.475.072	449,29	5,60	80,27	PC-Intel-I7 x GTX-680
Trabalho atual (4)	23.601.152	9203,63	89,43	102,91	PC-Intel-I7 x GTX-680

Tabela 27 – Comparação com *speedups* relatados na literatura

Onde: (1) corresponde a primeira implementação paralela/CUDA do 3D-Red-Black. (2) corresponde a otimização do modelo paralelo com a realocação dos processadores ociosos. (3) corresponde a otimização de alocação de threads via PSO e (4) corresponde a otimização de alocação de threads via PSO para nível de refinamento R1024. Os tempos exibidos na Tabela 27 referentes ao trabalho atual, correspondem a 2000 iterações.

Naturalmente, devido a diferenças entre métodos, plataformas e também alguma falta de detalhes relatados sobre implementações, uma comparação quantitativa seria pelo menos imprecisa (ou mesmo injusta); no entanto, algumas observações qualitativas poderiam ser feitas.

A primeira é que, mesmo usando computadores atuais, os tempos de execução de programas sequenciais podem atingir centenas de segundos, justificando as investigações e o desenvolvimento de novas abordagens em computação paralela.

Considerando o *speedup* alcançado por (SANJUAN, MARGALEF e CORTÉS, 2016), parece que a GPU é mais qualificada para a paralelização do Campo de Vento de diagnóstico do que um cluster de PCs.

Sobre a ordem de grandeza do *speedup* alcançado neste trabalho, pode ser considerado coerente com os resultados obtidos por Medina (MEDINA e CORTÉS, 2015). Note que, a GPU usada em (MEDINA e CORTÉS, 2015) é muito mais poderosa do que a usada aqui. Provavelmente, utilizar uma GTX-Titan na abordagem do presente trabalho poderia aumentar o *speedup* encontrado.

Como ocorre na presente investigação e no trabalho de Medina, o *speedup* também aumenta com o número de células no domínio computacional, apontando para uma boa escalabilidade.

Outra observação encontrada aqui foi que a otimização da quantidade de threads por blocos é uma tarefa um tanto quanto complexa, entretanto, faz-se extremamente necessária e o uso do algoritmo de otimização por enxame de partículas – PSO, é altamente recomendado.

CAPITULO 5. CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi investigado o refinamento do módulo de Campo de Vento do sistema de dispersão de radionuclídeos atmosféricos da CNAAA. São avaliados e analisados um critério de convergência refinado aplicado ao método de Minimização da Divergência e o refinamento do domínio computacional.

Observou-se que os critérios de parada usados no algoritmo de Minimização de Divergência era um número fixo de iterações. Devido a restrições computacionais na época em que o sistema foi desenvolvido (nos anos 80), apenas 56 iterações são feitas. Neste trabalho foi mostrado que muito mais iterações são necessárias de forma a alcançar $D \leq 10^{-3}$.

De fato, usando o domínio computacional original (R1), o impacto do refinamento no critério de convergência não é excelente em termos de tempo de execução absoluto, no entanto, demonstrou-se aqui que refinando o domínio computacional o tempo de execução atinge valores proibitivos para aplicações em tempo real.

Para superar várias dificuldades impostas pela natureza sequencial do método numérico original, permitindo uma eficiente paralelização do algoritmo, foi utilizada uma partição 3D-Red-Black para a decomposição do domínio. Comparações usando ventos reais observados demonstraram grande concordância com os resultados obtidos pelos algoritmos original e 3D-Red-Black.

Após ser verificada e testada a consistência do primeiro algoritmo, uma nova implementação foi desenvolvida com um melhor gerenciamento do uso de threads (eliminação dos threads ociosos). Observou-se uma melhora no tempo de execução do programa (*speedup*).

Enfim, uma terceira implementação do algoritmo foi iniciada. Desta vez, na versão paralela com a realocação dos processadores ociosos foi feita uma otimização da alocação de threads com o uso do algoritmo PSO, conseguindo-se *speedups* de mais de 50% no nível de refinamento R64 se comparado com a implementação paralela anterior.

Todas as simulações computacionais foram realizadas com diferentes níveis de refinamento (R1 a R64) e iterações. Foi observado um aumento do *speedup* à medida que o domínio computacional aumenta em todas as versões do algoritmo. Para o domínio original, foram obtidos *speedups* menos expressivos (cerca de 2,0), mas quando o domínio computacional aumenta, o *speedup* atinge valores interessantes. O tempo de execução caiu de 449,29 segundos (versão sequencial, executada em um PC Intel-I7) a 5,60 segundos (versão paralela e otimizada, executada em uma NVIDIA GTX-680), atingindo uma aceleração de 80,27 vezes. Na prática, acreditamos que este tempo de execução permitirá o uso do programa SCA refinado em tempo real.

Poucos trabalhos foram encontrados na literatura relacionada a abordagens paralelas para cálculos diagnósticos de Campos de Vento (SANJUAN, BRUN, *et al.*, 2014), (SANJUAN, MARGALEF e CORTÉS, 2015), (MEDINA e CORTÉS, 2015) e (SANJUAN, MARGALEF e CORTÉS, 2016)). Embora os métodos e plataformas computacionais sejam diferentes, dificultando comparações quantitativas justas, algumas observações podem ser feitas:

i) Mesmo usando computadores atuais, simulações de Campo de Vento de diagnóstico sequenciais podem exigir centenas de segundos para ser executado, justificando investigação e desenvolvimento de novas abordagens de computação paralela;

ii) Avaliando a aceleração alcançada por (SANJUAN, MARGALEF e CORTÉS, 2016), parece que a GPU é mais qualificada para a paralelização do Campo de Vento de diagnóstico do que clusters de PC. Além disso, os clusters são muito mais caros.

iii) Comparando o método proposto com outra abordagem de GPU (MEDINA e CORTÉS, 2015), a ordem de magnitude da aceleração alcançada neste trabalho (cerca de 80) é melhor que as obtidas por Medina (cerca de 40). Deve-se notar que a GPU usada no trabalho de Medina é muito mais poderosa do que a usada aqui. Provavelmente, usar uma GTX-Titan aumentaria o *speedup* alcançado neste trabalho. Poderia também ser observado no trabalho de Medina que quanto maior o domínio computacional, maiores são os resultados de aceleração, ratificando os resultados encontrados neste trabalho, e apontando para uma boa escalabilidade.

Com o presente trabalho, espera-se contribuir com mais um passo para melhorar o cálculo do Campo de Vento, permitindo que simulações mais refinadas sejam executadas em sistemas em tempo real. Além disso, o autor espera que os resultados aqui encontrados ajudem outras investigações futuras, permitindo aprimoramentos e comparações.

Num futuro próximo, espera-se que o programa de Campo de Vento baseado em GPU paralelo desenvolvido aqui seja integrado no sistema SCA da CNAAA. Isso ocorrerá assim que os outros módulos (Dispersão de Pluma e Projeção) tiverem suas versões paralelas implementadas.

De forma a estender os resultados obtidos neste trabalho e aperfeiçoar os métodos desenvolvidos, ficam como sugestões de trabalhos futuros:

- A utilização de outros algoritmos de otimização para a alocação dos threads por blocos;
- O uso de técnicas de compressão de matriz;
- O uso de matriciais esparsas;
- O uso de múltiplas GPUs em um cluster;
- Utilização das equações de Navier Stokes para determinação do campo de vento tridimensional não divergente.

REFERÊNCIAS

ALMEIDA, A. A. H. **Desenvolvimento de Algoritmos Baseados em GPU para Solução de Problemas na Área Nuclear**. Rio de Janeiro: PPGIEN/CNEN, 2009.

ARNOLD, D. et al. flexRISK – Flexible tools for assessment of nuclear risk in Europe. **Air Pollution Modeling and its Application XXI**, p. 737-740, 2012.

AUSTRALIAN GOVERNMENT. Evaluation of ARGOS for use in Australia. In: GRZECHNIK, M.; TINKER, R.; SOLOMON, S. **Technical Report No. 150**. [S.l.]: [s.n.], 2008.

BARTH, T. J.; CHAN, T. F.; TANG, W. P. A parallel non-overlapping domain-decomposition algorithm for compressible fluid flow problems on triangulated domains. **Contem. Math. 218**, p. 23–41, 1998.

BENI, G.; WANG, J. Swarm Intelligence in Cellular Robotic Systems. **Robots and Biological Systems: Towards a New Bionics?, Volume 102 of the series NATO ASI Series** , Tuscany, Italy, p. 703-712, 1989.

BROWN, M. J.; ARYA, S. P.; SNYDER, W. H. Vertical dispersion from surface and elevated release.An investigation of a non-Gaussian plume model. **Journal of Applied Meteorology**, p. 32,1263 –1280., 1993.

BUCK, I. **GPU computing with nVIDIA CUDA**. In: International. Conference on Computer Graphics and Interactive Techniques. ACM New York, NY, USA: [s.n.]. 2007.

CHRISTOUDIAS, T.; PROESTOS, Y.; LELIEVELD, J. Global risk from the atmospheric dispersion of radionuclides by nuclear power plant accidents in the coming decades. **Atmospheric Chemistry and Physics, 14**, p. 4607–4616, 2014.

CIMORELLI, A. J. et al. **AERMOD:** Description of Model Formulation. [S.l.]: Environmental Protection Agency - EPA, 2004.

COPPE/UFRJ - NUCLEAR, LABORATÓRIO DE ANÁLISE E SEGURANÇA. **SCA-MOD - Sistema de Controle Ambiental - Modelagem**. Rio de Janeiro: [s.n.], v. 3, 1987.

DANIELA MAIOLINO, N. S. **Modelo Termohidráulico para Realimentação do Cálculo de Seções de Choque Neutrônicas em Reatores PWR**. Rio de Janeiro: UFRJ/COPPE, 2011.

DE SAMPAIO, P. A. B.; JUNIOR, M. A. G.; LAPA, C. M. F. A CFD approach to the atmospheric dispersion of radionuclides in the vicinity of NPPs. **Nuclear Engineering and Design, 238**, p. 250–273, 2008.

DORIGO, M. **Optimization, Learning and Natural Algorithms**. [S.l.]: PhD thesis, Politecnico di Milano, 1992.

FABRICK, A. J.; SKLAREW, R. C.; WILSON, J. C. DEPICT, Formulation and User's Manual. **Science Applications, Inc. La Jolla, California, 92037**, 1976.

FABRICK, A.; SKLAREW, R.; WILSON, J. **Point Source Model Evaluation and Development Study**. [S.l.]. 1977.

FINARDI, S. et al. Evaluation of Different Wind Field Modeling Techniques for Wind Energy Applications Over Complex Topography. **Journal of Wind Engineering and Industrial Aerodynamics**, p. 74-76, pp. 283-294, 1998.

FLYNN, M. J. **Some computer organizations and their effectiveness**. [S.l.]: IEEE Transactions on Computers, v. 24, n. 9, 1972.

FORTHOFER, J. M.; SHANNON, K.; BUTLER, B. W. Simulating diurnally driven sIOPe winds with windninja. **In 8th Symposium on Fire and Forest Meteorological Society**, 2009.

FORTHOFER, J. M.; SHANNON, K.; BUTLER, B. W. Initialization of high resolution surface wind simulations using nws gridded data. **In Proceedings of 3rd Fire Behavior and Fuels Conference**, p. 25-29, 2010.

FREEMAN, T. L.; PHIILIPS, C. Parallel Numerical Algorithms. **Prentice Hall International**, 1992.

FURNAS CENTRAIS ELÉTRICAS S.A. **Especificação Funcional do Sistema de Controle Ambiental (SCA)**. Rio de Janeiro: [s.n.], 1987.

GLASKOWSKY, P. N. **White Paper NVIDIA's Fermi: the first complete GPU Computing Architecture**. [S.l.]. 2009.

GOLDBERG, D. E.; HOLLAND, J. H. Genetic Algorithms and Machine Learning. **Machine Learning V.3**, p. 95-99, 1988.

GRIMMOND, C. S. B.; OKE, T. R. Aerodynamic properties of urban areas derived from analysis of surface form. **Journal of Applied Meteorology. v. 38**, p. 12621292, 1999.

HALL, D. J. et al. The UDM: A Puff Model for Estimating Dispersion in Urban Areas. **7th Int. Conf. on Harmonisation within Atmospheric Dispersion in Urban Areas**, p. 256 - 260.

HARVEY, P.; HAMEED, S.; VANDERBAUWHEDE, W. Accelerating Lagrangian particle dispersion in the atmosphere with OpenCL across multiple platforms. **IWOCL**

'14 Proceedings of the International Workshop on OpenCL 2013 & 2014, Bristol, United Kingdom, v. Article No. 6, 12 maio 2014.

HEIMLICH, A.; MOL, A. C. A.; PEREIRA, C. M. N. A. GPU-based Monte Carlo simulation in neutron transport and finite differences heat equation evaluation. **Progress in Nuclear Energy (New Series), 53**, p. 229-239, 2011.

HEIMLICH, A.; SILVA, F. C.; MARTINEZ, A. S. Parallel GPU implementation of PWR reactor burnup. **Annals of Nuclear Energy, 91**, p. 135-141, 2016.

HOMICZ, G. F. **Three-Dimensional Wind Field Modeling: A Review - SAND REPORT: SAND2002-2597.** [S.l.]. 2002.

HUANG, T.-C.; HSU, P.-H. A practical run-time technique for exploiting loop-level parallelism. **Journal of Systems and Software, 54**, p. 259-271, 2000.

HUSEMANN, R. et al. **Evaluation of CUDA GPU Architecture as H.264 Intra Coding Acceleration Engine**. Salvador: [s.n.], 2013.

HWU, W.-M.; KEUTZER, K.; MATTSON, T. G. The Concurrency Challenge. **IEEE Design and Test Computers**, p. 312-320, 2008.

INTERNATIONAL ATOMIC ENERGY AGENCY. **Energy, Electricity and Nuclear Power estimates for the period up to 2050**. Vienna, p. 58. 2013. (978-92-0-11191 0-0).

KENNEDY, J.; EBERHART, R. **Particle Swarm Optimization**. IEEE International Conference on Neural Networks. IV. [S.l.]: [s.n.]. 1995. p. 1942-1948.

KOVALETSA, I. V. et al. Influence of the diagnostic wind field model on the results of calculation of the microscale atmospheric dispersion in moderately complex terrain. **Atmospheric Environment. DOI: 10.1016/j.atmosenv.2013.06.015**, p. 79:29-35, 2013.

LAUB, A. J. A schur method for solving algebraic riccati equations, v. 24, n. 6, p. 913–921, December 1979.

LELIEVELD, J.; KUNKEL, D.; LAWRENCE, M. G. Global risk of radioactive fallout after major nuclear reactor accidents. **Atmospheric Chemistry and Physics, 12**, p. 4245–4258, 2012.

LIOU, K. N. **An Introduction to atmospheric radiation**. New York: Academic Press, 1980.

LONGHETTO, A. **Atmospheric planetary boundary layer physics:** proceedings of the 4th course of the International School of Atmospheric Physics. Developments in atmospheric science (vol. 11). Amsterdam: Elsevier. 1980.

MEDEIROS, J. A. C. C.; SCHIRRU, R. Identification of Nuclear Power Plant Transients using the Particle Swarm Optimization algorithm. **Annalsof Nuclear Energy, 35**, p. 576-582, 2008.

MEDINA, C. T.; CORTÉS, F. A. Reducing runtime of WindNinja's wind fields using accelerators TFG en Enginyeria Informatica. **Universitat Autònoma de Barcelona. Escola d'Enginyeria**, 30 junho 2015. Disponivel em: <http://ddd.uab.cat/record/143558>.

MELO, A. M. V. **Avaliação de Desempenho dos Modelos AERMOD e CALPUFF Associados ao Modelo PRIME**. [S.l.]: Dissertação de Mestrado - Universidade Federal do Espírito Santo., 2011.

MENESES, A. A. M.; MACHADO, M. D.; SCHIRRU, R. Particle Swarm Optimization applied to the nuclear reload problem of a Pressurized Water Reactor. **Progress in Nuclear Energy, 51**, p. 319-326, 2008.

MORAES, M. R. **Implementação de um sistema de modelos para a qualidade do ar**. Florianópolis: Dissertação (Mestrado em Engenharia Mecânica). Universidade Federal de Santa Catarina, 2001.

MORAES, S. R. D. S. **Computação Paralela em Cluster de GPU Aplicado a Problema da Engenharia Nuclear**. Rio de Janeiro: CNEN/IEN, 2012.

NICOLAU, A. D. S. **Algoritmo Evolucionário de Inspiração Quântica Aplicado na Otimização de Problemas da Engenharia Nuclear**. Rio de Janeiro: UFRJ/COPPE, 2014.

NUCLEAR REGULATORY COMMISSION. **RASCAL 4: Description of Models and Methods**. [S.l.]. 2012.

NVIDIA CORPORATION. **NVIDIA CUDA Compute Unified Device**. [S.l.]. 2008a.

NVIDIA CORPORATION. **NVIDIA GeForce GTX 200 GPU Architectural**. [S.l.]. 2008b.

NVIDIA CORPORATION. **GTX 680 Kepler Whitepaper**. [S.l.]. 2012.

NVIDIA CORPORATION. **CUDA C Programming Guide**. [S.l.]. 2014.

NVIDIA CORPORATION. Fast N-Body Simulation with CUDA. In: NYLAND, L.; HARRIS, M.; PRINS, J. **GPU Gems 3**. [S.l.]: NVIDIA Corporation, 2015. p. 677 - 695.

NVIDIA CORPORATION. **CUDA C Programming Guide**. [S.l.]. 2017.

OKE, T. R. **Boundary layer climates**. London: New York: Routledge, 1978.

OLIVEIRA, I. M. S. D. **Inteligência de Enxames Aplicada ao Problema de Otimização de Recargas de Reatores Nucleares a Água Pressurizada**. Rio de Janeiro: UFRJ/COPPE, 2013.

PARHAMI, B. **Introduction to Parallel Processing:** Algorithms and Architectures. New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers, v. Series in Computer Science, 2002.

PASQUILL, F. The estimation of the dispersion of windborne material. **Meteor. Mag. 90**, p. 33-49, 1961.

PEDERSEN, U.; LEACH, S.; HANSEN, J.-E. S. **Biological Incident Response:** Assessment of Airborne Dispersion. Denmark: [s.n.], 2007.

PEREIRA, C. M. N. A. et al. Development and performance analysis of a parallel Monte Carlo neutron transport simulation program for GPU-Cluster using MPI and CUDA technologies. **Progress in Nuclear Energy (New Series), 65**, p. 88-94, 2013.

PEREIRA, C. M. N. A.; SACCO, W. F. A parallel genetic algorithm with niching technique applied to a nuclear reactor core design optimization problem. **Progress in Nuclear Energy (New Series) 50**, p. 740-746, 2008.

PHAM, D. T. et al. **The Bees Algorithm – A Novel Tool for Complex Optimisation Problems**. Cardiff: Manufacturing Engineering Centre, Cardiff University, 2006.

PINHEIRO, A. et al. GPU-Based Parallel Computation in Real-Time Modeling of Atmospheric Radionuclide Dispersion. **Advances in Human Factors and System Interactions**, Florida, USA, v. 497, p. 323-333, 27-31 Julho 2016. ISSN Part V. Disponivel em: <http://link.springer.com/chapter/10.1007/978-3-319-41956-5_29>.

RENTAI, Y. Atmospheric Dispersion of Radioactive Material in Radiological Risk Assessment and Emergency Response. **Progress in Nuclear Science and Technology, Vol. 1**, p. 7-13, 2011.

ROCHA, J. M. L. D. A. **Aceleração GPU da Animação de Superfícies Deformáveis**. Lisboa: [s.n.], 2008.

RYOO, S. et al. **Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA**. PPoPP'08 - Proceedings of the 2008 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Salt Lake City, UT, United States: [s.n.]. 2008. p. 73-82.

SANJUAN, G. et al. **Determining map partitioning to accelerate wind field calculation**. 2014 International Conference on High Performance Computing & Simulation (HPCS). Bologna: [s.n.]. 2014. p. 96–103.

SANJUAN, G.; MARGALEF, T.; CORTÉS, A. Adapting Map Resolution to Accomplish Execution Time Constraints in Wind Field Calculation. **Procedia Computer Science, Volume 51**, p. 2749-2753, 2015.

SANJUAN, G.; MARGALEF, T.; CORTÉS, A. Applying domain decomposition to wind field calculation, v. 57, p. 185-196, September 2016.

SCIRE, J. S.; STRIMAITIS, D. G.; YAMARTINO, R. J. **A user's guide for the CALPUFF dispersion model (Version 5)**. Concord, MA: Earth Tech. Inc., 2000.

SEINFELD, J. H.; PANDIS, S. N. **Atmospheric Chemistry and Physics, From Air Pollution to Climate Change**. New Jersey: John Wiley & Sons, Inc., 2006.

SINGH, B. et al. Accelerating urban fast response Lagrangian dispersion simulations using inexpensive graphics processor parallelism. **Environmental Modelling & Software**, v. 26, n. 6, p. 739-750, 2011.

SOARES, M. S. **Avaliação do Sistema de Modelagem CALPUFF aplicado ao dióxido de enxofre para as Bacias Aéreas I, II e III da Região Metropolitana do Rio de Janeiro**. Rio de Janeiro: Monografia (Graduação em Meteorologia) - Universidade Federal do Rio de Janeiro, 2010.

STULL, R. B. **An introduction to boundary layer meteorology**. Dordrecht: Kluwer Academic Publishers, 1988.

TILL, J. E.; GROGAN, H. A. **Radiological Risk Assessment and Environmental Analysis**. New York: Oxford University Press, 2008.

TRELEA, I. O. The particle swarm optimization algorithm: convergence analysis and parameter selection. **Information Processing Letters, 85**, p. 317-325, 2003.

VOLKOV, V.; DEMMEL, J. W. **Benchmarking GPUs to Tune Dense Linear Algebra**. SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. [S.l.]: IEEE. 2008.

WAINTRAUB, M.; SCHIRRU, R.; PEREIRA, C. M. N. A. Multiprocessor modeling of parallel particle swarm optimization applied to nuclear engineering problems. **Progress in Nuclear Energy (New Series) 51**, p. 680-688, 2009.

WHITEHEAD, N.; FIT-FLOREA, A. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. **rn (A + B), 21:1-1874919424.**, 2011.

WOLFE, M. J. **High Performance Compilers for Parallel Computing**. [S.l.]: Addison Wesley Publishing Company Incorporated, 1996.

WONG, H. et al. **Demystifying GPU Microarchitecture through Microbenchmarking.** IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). [S.l.]: [s.n.]. 2010.

YANG, D.-X. et al. Theory on AERMOD Modeling System. **Chemical Industry and Engineering**, 2005.

YAVNEH, I. R. On Red-Black SOR Smoothing in Multigrid. **SIAM J. Sci. Comput., 17(1)**, p. 180–192, 1995.

ZANNETTI, P. **Air Pollution Modeling**. Southampton: Springer, 1990.